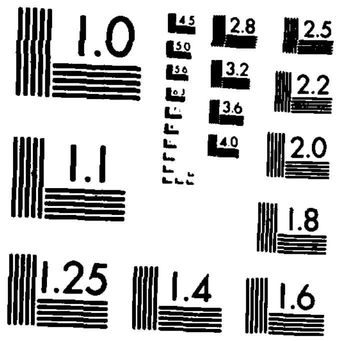
AD-A125 441 COMMUNICATION ENVIRONMENTS FOR LOCAL NETWORKS(U) 1/3
UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY
INFORMATION SCIENCES INST P V MOCKAPETRIS DEC 82
UNCLASSIFIED ISI/RR-82-103 MDA903-81-C-0335 F/G 17/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ISI/RR-82-103

December 1982

Paul V. Mockapetris

Communication Environments for Local Networks

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

ISI

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-82-103	2. GOVT ACCESSION NO. AD-A125 441	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Communication Environments for Local Networks		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) Paul V. Mockapetris		8. CONTRACT OR GRANT NUMBER(*s*) MDA903-81-C-0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE December 1982
		13. NUMBER OF PAGES 198
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) ----------		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

----

18. SUPPLEMENTARY NOTES

In a slightly different form, this report was the author's PhD thesis at the University of California, Irvine.

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

broadcast, communication protocols, communication services, front end processor, Internet Protocol, local network environment, local networks, multicast, network interface, protocols, Transmission Control Protocol

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

20. ABSTRACT

This report studies the problem of providing high performance communication services to computers on a local network, or other high-speed network, without sacrificing compatibility with existing communication protocols. The focus of the investigation is the network interface, the I/O controller which interfaces computers to the network medium. The network interface proposed in this report contains a novel processing element called the *filter*, in addition to a fairly common microprocessor-based architecture. The filter is a high-speed custom processor and a memory which holds programmable descriptions of packet formats and connection records for the protocols supported by the interface. The filter could be implemented using VLSI or a small amount of readily available bit-slice components. The filter enables the interface to process most transmissions as they arrive on the communications medium. Filter processing adds negligible delay to the overall communication path and, in most cases, replaces processing which is conventionally performed in the software of the interface microprocessor or the host. Thus, communications systems built with the new interface are limited by the speed of the medium and not by processing delays. The filter interface is shown to be practical for a 10 Mbps medium and several protocols, including the Internet Protocol (IP) and the Transmission Control Protocol (TCP). The report also considers additional methods for improving communications which are only applicable to the local network environment. Reliable multicast and special acknowledgment mechanisms are considered. These appear to be of significant value, especially when combined with the high performance of the filter interface.

ISI/RR-82-103
December 1982

Paul V. Mockapetris

Communication Environments for Local Networks

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH WAS SUPPORTED BY THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO MDA903-81-C-0335 AND BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT NO GJ 1045 VIEWS AND CONCLUSIONS CONTAINED IN THIS REPORT ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF DARPA, NSF, THE U.S. GOVERNMENT, OR ANY PERSON OR AGENCY CONNECTED WITH THEM

ACKNOWLEDGMENTS

I wish to thank my thesis committee for their patience and efforts on my behalf. I am especially grateful to Stephen D. Crocker for his accurate criticisms of several versions of this report.

I am grateful to all the people in the ICS department at UC Irvine who helped me during my graduate studies, and especially the participants in the DCS project who listened to many of my inchoate ideas, and helped me refine them.

I owe a great deal to Jon Postel of the USC Information Sciences Institute for his understanding and for educating me about large scale networks. Thanks are also due to my back hall cohorts at ISI who helped me in many ways.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

This report provides insight into several problems in the design, implementation, and use of computer networks. The focus of this work is the creation of communication services for a local network environment. Contemporary implementations of these services include standard protocols for compatibility and separate efficient protocols for applications which need the high bandwidth typical of local networks. This report contains an interface design that allows very efficient implementations of standard protocols as well as some valuable extensions.

This chapter discusses the relationship between local networks and other forms of multiple computer systems, and introduces the organization and methods of this report.

1.1 PROBLEM AREA DESCRIPTION

Multicomputer Systems

Many interrelated motivations exist for the work that has been done on computer networks, multiprocessors, and other forms of multiple computer systems. One way to classify these motivations is as follows:

1. (Performance) We may want to apply multiple computing elements to the same problem in order to have more power available for solving the problem.
2. (Reliability) We may want to use multiple independent resources to avoid dependence on a single resource.
3. (Conformance to problem structure) The structure of the problem may necessitate multiple resources, or be more conveniently represented by multiple components. We often need to apply processing power in a manner that parallels the distributed nature of our institutions. (e.g., businesses, government, research communities.)
4. (General communication services) The system may be supporting services such as mail and file transfer, in which the communication facilities are not really used to link parts of a single computation, but rather as a "common carrier" to move data between users.

One reason for the large number of different multiple machine structures is that they are designed on the basis of different combinations of these considerations. Other factors include interconnection technologies and requirements such as bandwidth, delay, and the size of the area to be served. The following are some representative systems:

Array processors, such as the ILLIAC 4 , and vector processors, such as the CRAY-1 and CYBER 205, organize processors in a matrix and are successful in providing increased throughput for problems that can be conveniently stated in the appropriate form [Levine 82]. The interconnection in these systems has the high bandwidth and parallelism typical of tightly coupled designs, but is often restricted in connectivity. For example, an ILLIAC 4 processor may only communicate with its immediate neighbors, or with processors in the same row or column of the processor array. The motivation for these systems is primarily (1), that is, increased throughput.

More general forms of multiprocessor systems allow for more flexible connectivity, and may also add processing power in the basic communication path. The Cm* system [Fuller 78] is a sophisticated member of this group. Such systems are still performance oriented, but also have elements of the other motivations.

Multiprocessors are built in redundant configurations to provide ultra-reliable services. The interconnection is tightly coupled, and contains logic for error detection and component selection. The Pluribus [Heart 73] and ESS telephone switching systems [BSTJ 81] are examples of the use of multiprocessor systems to provide very reliable communication-related computer services. Other examples are Tandem financial systems [Katzman 78] and systems for spacecraft. The high cost of these systems is justified by the higher cost of failure, and hence these systems are based mainly on the reliability motivation.

Systems motivated by the desire to conform to a particular problem structure are found in process control [Sherman 78, Smith 79] and other forms of distributed systems. One study [Holmgren 79] calls for local networks to be used to link the various processors and devices in a military command center. The resultant structure is similar to that found in bus-oriented minicomputers, except that the network is used for data transfer between otherwise independent systems. The interconnection schemes used in these systems are varied. A study [CSDL 70] evaluating experiences with the APOLLO guidance computer used for the moon missions recommended that several decentralized processors would be a better choice for the space shuttle than the single computer system used in APOLLO, even ignoring the reliability enhancement of multiple processors, due to the lower overall complexity and improved real-time response.

Digital communication systems have been built purely to implement general communication services, such as computer mail. The economics of packet switching suggests that the future may see these systems in areas traditionally implemented with circuit switching, such as voice [Roberts 78].

Packet-switched networks, such as the ARPANET [Roberts 70, McQuillan 77], are typically based on lower speed leased lines, and use a great deal of processing power in the communication system to use the limited bandwidth efficiently, to create the illusion of complete connectivity, and to otherwise enhance the properties of the media. These systems are composed of loosely coupled hosts which contain the majority of the protocol implementation. The user community of these systems is large enough and diverse enough so that any of the previously mentioned motivations are valid in some cases, though the primary use of these systems is for general communication services.

Networks

Networks are organized in terms of standards (protocols) which attempt to define precisely the rules for interactions among parts of the network while leaving other network parameters as unrestricted as possible. The intent is to guarantee that communication can take place as long as the standards are observed, while allowing different hosts, operating systems, applications, and modes of interaction.

Each protocol transforms the resources available to it into a new, more powerful set of resources. Protocols for the lowest level deal with actual hardware and operating system resources; intermediate level protocols build on the resources of underlying layers; the top level presents resources to the user or a user program.

In general, the base-level network is modelled as a graph. Called hosts, the nodes on the graph are computer systems which perform functions for the user. The arcs of the graph are communication lines. These communication paths may be physical links, shared buffers, or even a complex of communication processors and links. The common factor is that these paths are all treated as if they were simple serial links. This abstraction, often called "loose coupling" or "thin wire," guarantees that any type of communication path could be used, regardless of its width or other properties.

The lowest level protocol, called the link level, defines procedures for transmitting an ordered group of bits (a packet) over a particular communication link as a unit. A host may well have several communication links, and hence several link-level protocols.

Local Networks

Local networks are systems that provide service within a limited geographical area; they usually have a radius of a mile or so. A local network spans a building, campus, research park, or other similarly sized facility, and is owned by the corresponding entity.

Because of the limited distances, local networks use privately owned transmission media such as twisted pair, coaxial cable or fiber optic links. Although more exotic and expensive systems achieve rates in the 50-100 Mbps range, a typical local network has a transmission rate in the 1-10 Mbps range, using one of these media, baseband modulation, and inexpensive components. This speed is far above that used in long haul networks such as the ARPANET, where 50 Kbps links are typical, and is within an order of magnitude of the raw memory bandwidth seen in minicomputers. For example, the PDP-11 UNIBUS, operating in word mode with 330 ns memory cycles, has about 50 Mbps of bandwidth.

The cost of interfacing a host to a local network is typically much lower than that of interfacing to a long haul network. To illustrate the point, connection to the ARPANET requires the use of an Interface Message Processor (IMP) or a Terminal Interface Processor (TIP) costing between $50K and $100K. This high cost is due to the highly sophisticated minicomputer system included in the IMP or TIP. Interfaces to local networks are at least an order of magnitude cheaper, and are comparable in complexity to sophisticated peripheral interfaces [Mockapetris 77, Carpenter 78].

Local network technology retains this cost advantage even compared to a single message switch shared by all of the hosts of a network. In a MITRE study [Holmgren 79], the total costs of a system based on a central switch were compared to that of a "cable bus" system based on local network technology. The comparison assumed a 10 year life cycle and included all recurring and non-recurring costs due to the communication system and host interfaces. The cable bus was predicted to be 27 percent less expensive. If a reliable configuration was constructed in which critical components were duplicated, the cable bus savings increased to 51 percent.

Thus local networks have a unique set of strengths:

1. The high bandwidth of the communication medium places few restrictions on system architecture. Local network media can support rates of interaction similar to those found in tightly coupled multiprocessor systems. Systems which combine high bandwidth with the loosely coupled organization typical of networks offer the best parts of both worlds. Such a hybrid could exhibit the modularity, extensibility, defensiveness, and organizational flexibility of a network and the intimate interface between communication and computation of a multiprocessor.

2. Because a local network is owned and controlled by its users, any part of the network architecture can be tailored to fit the users' needs. The network need not adhere to external standards unless there is some benefit for doing so.
3. The simplicity of access to the local network means that minicomputers and other less capable hosts can be added to the network. This is important in light of the trend to embed processors as smart controllers in terminals, instruments, and the like. Connection to the network means that the data generated by these devices can be captured at the source. Network access can also be used to enhance these devices' capabilities through the use of resources delivered through the network.

Current local networks capitalize on these strengths, although the benefits of the local network environment are offset somewhat by structures inherited from long haul networks. The inherited structures, which were optimal for the long haul environment, do not allow full utilization of local network capabilities. Two local networks which illustrate differing approaches are the Ethernet and the DCS.

The Ethernet [Metcalfe 76] is composed of a contention medium and minicomputer hosts. It uses a layered protocol system [Boggs 79] very similar to that found in long haul networks. Multiple Ethernets be be connected to form a composite network. The resultant system is no longer local in the geographical sense, and typically includes low-bandwidth telephone links which transfer packets between Ethernets.

The Distributed Computer System (DCS) [Farber 73, Mockapetris 79] is a local network which integrates communication into the basic system design. Message arrivals drive process execution in a manner similar to the way tokens drive Petri nets [Petri 62]. User processes see a single level of communication protocol, which the processes access through simple send and receive primitives. Because messages are the only form of interprocess communication, the distribution of processes among DCS hosts is never limited by the pattern of process interaction.

These systems are successful, but their expansion and refinement are hampered by protocol implementation structures inherited from long haul networks.

Protocol implementations consume resources such as memory space, CPU time, and operating system services. The programs which implement the protocol must be brought up on every machine type on the network. We would like to satisfy these needs in a low cost network interface. The ideal interface implements all of the communication environment without consuming any host resources.

These systems fail to transform the high bandwidth of the medium into a high data rate for process-to-process connections. This is especially true when using standard protocols which provide compatibility with other hosts and networks. The problem becomes acute if we wish to add graphics, speech, facsimile, voice or other high bandwidth applications to our systems. Part of the problem is the time it takes for communication to percolate through the multiple layers of protocol software. The other part of the performance problem is inherent in the operating system of the host: most systems take several milliseconds to switch contexts, and protocol implementations can often require more than one context switch per processing action. The designer has two unpleasant choices: accept the low bandwidth of available standard protocols, or use an efficient but incompatible protocol.

Layered protocols also fail to transfer low-level capabilities to all higher levels. For example, most media have arbitration conventions which aren't made available to higher layers. Low-level acknowledgment systems are another example. In a sense, this is intentional. The layered approach is the natural method for building ever more powerful and complicated levels of abstraction from very simple base primitives provided by the network interface. While this means that we can build a very powerful protocol on top of a "stupid" interface, it usually means that we can't use the added capabilities of a "smart" interface. As a consequence, low-level facilities are often constructed in a manner appropriate for the next level of protocol, rather than a higher or ultimate layer. For example, low-level acknowledgments are usually designed with the link layer of software in mind, rather than the transport layer.

This Report's Approach

We should not discard structured design, but we should augment the criteria we use to modularize the communication environment. Current modularizations are based on the use of absolutely minimal interface capabilities, protocols residing in the network hosts, and the desire to reduce the intellectual difficulty of implementing the protocol software.

A modularization appropriate to the local network environment should exhibit "horizontal" as well as the usual "vertical" protocol layers. That is, the conventional layers should be further separated according to function, so that a local network medium that implements a particular facility can pass the benefits of the facility up to higher layers, bypassing the usual synthesis process for that facility used in less capable interfaces.

For example, broadcast and arbitration functions are usually developed at a high level in the protocol hierarchy, even though similar facilities are available at the lowest level of the network. These low-level facilities can't be easily added to existing protocol implementations because the applicable abstractions don't exist in the lower protocol layers.

Redundant implementations of some facilities are essential to improve performance (the "best efforts" principle cited in [Metcalfe 73]) while preserving reliability ("end-to-end" error correction). For example, low-level acknowledgments may be able sufficient for 99.99 percent of all acknowledgments. If a low-level acknowledgment facility exists in parallel with a conventional acknowledgment facility, the system can avoid the overhead of transmitting the software acknowledgment in 99.99 percent of the cases, thus improving performance. The redundant system does not depend on the low-level acknowledgment system for the last .01 percent of the cases, and thus has the same reliability as a pure software "end-to-end" system.

The goal of this report is to explore the construction of these new environments. All of the communication environment is divided into three parts, with questions to be examined in each part:

1. *The media interface and signalling conventions.* What can we assume about the communication medium? How can we take advantage of the communication medium's properties?
2. *The protocol.* How do we map between the user interface and the medium interface? Can we build high-performance versions of protocols originally designed for compatibility?
3. *The user interface.* How does the user process describe the services that it wants the communication environment to perform? What objects and commands are required? How should a standard interface operate with different host environments?

In contemporary long-haul networks, the network interface is a simple I/O interface, the user interface is implemented totally in host software, and all, or at least the majority, of the protocols are implemented in the host. The total communication environment is supported by a separate body of software in the switching nodes.

This report is based on the premise that as much as possible of all three of these parts should be moved from the host into the communication interface. This method should yield the following advantages:

1. Offloading the host should improve host performance and possibly simplify the task of interfacing to a new host or network.
2. If the communication interface is tailored to the job of protocol execution, protocol processing can be done at a rate approximating the data rate of the medium. If the network interface has a high-bandwidth path to the host, the raw bandwidth of the medium will be delivered to the user in the form of usable bandwidth. We will reduce the net end-to-end delay for communication between processes, and not just move the delay out of the host into a front end.
3. New services, such as reliable broadcast, can be incorporated as primitives.

In order to achieve these goals, the three parts of the communication environment must be designed to fit together.

The media interface should provide as much support as possible to protocol mechanisms.

The protocol mechanisms must provide an efficient translation between user and media interfaces. The structure of the algorithms and the computing resources is critical; we may wish to separate various protocol functions with horizontal, as well as the usual vertical, partitions. This enables a smart interfaces to participate in tasks that should not be isolated in a single protocol layer. The partitioning discipline should seek functional distinctions and avoid distinctions that are based on a particular technology (e.g., rings, fiber optics, Ethernets).

The user interface must define services that carry the power of the media, while at the same time providing a firm specification of a service that will fit in the interface.

1.2 SIGNIFICANCE OF PROBLEM

The significance of this area, and hence this report, is based on the following chain of reasoning:

1. (Traditional Networks) Traditional long-haul computer networking is valuable.
2. (Local Networks) Local networks are valuable for the same reasons as long-haul networks, as well as others.
3. (New Methodologies) A search for new methodologies, tailored to local networks, is justified.

Traditional Networks

Today the most significant form of computer networking is based on packet-switching technology. Packet networks were first proposed by Paul Baran in a 1964 Rand report on new technologies for U.S. Air Force communication [Baran 64]. Actual network designs were published in the late 1960s, including the ARPANET [Roberts 67] and NPL networks [Davies 67]. A minimal ARPANET, including 4 hosts, became operational in 1969.

The 1970s saw the ARPANET increase in size to approximately 200 nodes of various types [NIC47000 79]. In foreign nations, government postal and telecommunication monopolies have moved into the field with common carrier systems in Canada [Clipsham 76, Datapac 76], the U.K. [Belton 74], France [Danet 76], and elsewhere. The U.S. has seen several commercial networks [Telenet, Tymes 71] and awaits the completion of the Bell ACS system, and IBM's Satellite Business Systems (SBS) network.

Packet-switched networks seem certain to enjoy an expanding volume in their present applications, and seem likely to make inroads in areas traditionally dominated by circuit-switching technology. New modes of communication, such as computerized mail, are certain to grow in use and complexity [Lederberg 78]. These new forms of use are based on packet switching.

Local Networks

As previously mentioned, local networks are not necessarily restricted to the same set of applications as long-haul networks. They present a level of performance and capability that make new services possible. Shared file systems [Swinehart 79] and subsystems constructed out of multiple hosts are examples.

At the same time, the potential local network owner may wish to provide communication <u>within itself</u> that long-haul networks provide to a larger community. Although traditional communication methods (hand carry, memos, etc.) are more competitive in the local environment, we would expect to see more possibilities for local use simply because the internal data flow is larger. Local networks must make communication easier than before if they are to be used for this function. Being able to easily interface to the local network is easily as important as high bandwidth.

Internal networks have been developed by many organizations [Metcalfe 73, Carpenter 78, Gerhardstein 78]. Often these networks grow until they are no longer geographically local, and include other communication media. In order to allow for subsets of users to continue to have the benefits of the local network environment, these large systems can be partitioned into clusters [Boggs 79, Clark 78].

New Methodologies

Given that local networks are desirable, why do we need new techniques? Why not continue to use the same technology as large scale networks? The answer is that long-haul network technology can't deliver the performance, functionality, or low cost typical of local networks.

Communication is a field that, by definition, interfaces different systems. As such, we expect a multitude of tradeoffs. Different situations call for different choices. This view is justified by past experiences in this area. Packet transmission became effective only after the cost of processing power dropped [Roberts 78]. The ARPANET protocols were unacceptable when used on high-bandwidth, high-latency satellite links [Kleinrock 78]. The claim has even been made that virtual circuits will eventually triumph over datagrams due to the dropping cost of memory [Roberts 78].

While history supports the view that the form of a communication environment's implementation should be matched to the user's needs and the form of the transmission system, it also suggests that any such efforts may themselves be made obsolete by new developments. Thus it is important to note that local ownership, high performance, and the low relative cost of bandwidth are not tied to a particular technology. Approaches for dealing with today's local networks will be applicable to other systems which share these properties.

1.3 THEORETICAL FRAMEWORK

The scope of this report is limited by the following assumptions and restrictions:

1. The communication environment should be primarily supported in an autonomous interface outside of the host. This choice allows specialized architectures, offloads the processing requirements, and eliminates the need to code network software for every host type.
2. The target communication environment should be at least as rich as that provided by TCP or the DCS system SEND and RECV primitives. TCP is a standard protocol; the DCS system is representative of a simple message system.
3. Extensions should follow familiar abstractions where possible.
4. The choice of transmission media will be restricted to either a ring, a contention system (such as Ethernet), or simple extensions to these systems. These are the most cost effective choices available today.
5. Media bandwidth is available in abundance; hence it will be used up to provide more functionality when possible.
6. The host hardware interface is assumed to be a high-bandwidth system, such as DMA or a channel interface. This is necessary to match the arrival rate of data from the medium.
7. The host is assumed to have a byte-addressable memory. This choice is not essential, but simplifies discussion.

1.4 GENERAL OUTLINE AND PLAN OF REPORT

This report is organized into three phases:

1. Survey of current state of the art (Chapters 2-3)
2. Model evaluation (Chapters 4-6)
3. Summary and conclusions (Chapter 7)

The first phase of the report surveys the current state of the art in local networks and communication protocols. In keeping with the previously mentioned model of the communication environment as having three parts (media interface, protocol mechanism, user interface), this phase looks at models for all three. One purpose of this phase is to enumerate the individual protocol mechanisms used to build the communication environment, problem areas in local network design, and opportunities for enhancement.

The second phase of the report begins in Chapter 4 by outlining the components of a model interface. The next two chapters apply the model to tasks which are driven by the arrival rate of the medium and tasks which are performed independently or arriving data.

The last phase evaluates the rest of the report, and suggests directions for future study.

Chapter Outlines

Chapter 2 - Current Local Network Systems

Chapter 2 examines the current state of the art in terms of local networking media and interfacing techniques. Ring and bus systems are carefully examined, and large-scale network and multiprocessor techniques are used for comparative purposes.

Chapter 3 - Contemporary Communication Environments

This chapter defines some salient features of protocols, and studies the parts of two communications environments in depth. The DCS system is studied as a representative of local network protocols that stress ease of use and simplicity. TCP is studied as a representative of protocols that allow compatible transfers across varied networks, hosts, and operating systems.

Chapter 4 - A Model Architecture

This chapter introduces an interface architecture that is refined in later chapters. The architecture is discussed in terms of the goals and available implementation structures.

Chapter 5 - Message Binding

Chapter 5 studies algorithms for processing packets as they arrive on the medium. This activity, called binding, locates the connection record for an arriving packet, processes packet control fields, calculates the parameters used to control data transfer, generates a prompt acknowledgment, and updates connection state.

Chapter 6 - Data Delivery

Chapter 6 considers problems related to the management of communicated data. The principal issues are the design of data buffering to prevent the interface from becoming a bottleneck in the data transfer path, and the design of control signals which allow the network interface and the host to coordinate the progress and completion of user requests.

Chapter 7 - Summary and Conclusions

This chapter presents the overall findings of this report, and suggests strategies to be used and directions for future work.

2. CURRENT LOCAL NETWORK SYSTEMS

2.1 INTRODUCTION

A communication system is the part of the total communication environment that is external to the hosts of the network. This communication system has two major parts: the medium which carries signals between host sites, and the network interfaces which connect hosts to the medium.

Designers of computer interconnection systems have developed a large number of architectures for the communication system. The large number of architectures is in part due to the large number of applicable technologies for media and interfaces, and in part due to design issues related to the higher order parts of the communication environment. In most cases the architectures chosen for local networks are those which combine simplicity, and hence low cost, with a structure tailored to the local network environment.

Anderson and Jensen proposed a taxonomy for all forms of computer interconnection structures which helps to illustrate the types of systems used in local networks [Anderson 75]. The taxonomy uses the following characteristics to separate classes:

1. Transfer strategy (direct vs. indirect)
2. Transfer control method (centralized vs. decentralized)
3. Transfer path structure (shared vs. dedicated)
4. System architecture

The first test is transfer strategy. In indirect transfer, messages are routed through switching entities that process the message. In direct transfer, messages pass only through the medium on their way between hosts.

Repeaters and similar components are considered to be part of the medium. Routing, protocol translation, and similar facilities denote indirect transfer; often the components that perform indirect transfer are indistinguishable from normal hosts, or are processes in hosts.

Local networks aren't compelled to use indirect transfer to be compatible with external standards, nor are they interested in using such methods to increase medium utilization. In the absence of other factors, the simplicity of direct transfer is attractive. Thus most local networks use direct transfer.

The second level of the taxonomy is the transfer control method. This refers to the choice between centralized and distributed control of the functions of indirect transfer. In direct transfer systems, the choice is moot: no control is necessary.

The third level of the taxonomy distinguishes between systems that share the use of communication paths and those that do not. A dedicated path carries data from one point to another; such a path is often termed point-to-point. In general, a point-to-point full-duplex path is considered to be two dedicated paths. A shared path has multiple sources or destinations; multidrop or bus media are shared.

Assuming that the connectivity is uniform and not redundant, these considerations lead to two topologies: a circular system, or a bus system. These two geometries are the most popular for local networks because they require the minimum number of connections, hence the minimum amount of hardware.

The last level of the taxonomy is system architecture. This level separates implementation structures which are identical with respect to the first three criteria.

Using these four levels, the taxonomy enumerates 10 categories that represent existing systems. The circular and bus systems used in local networks are covered by 3 of these categories: circular systems with and without central control, and bus-like systems.

In a bus system, all of the hosts are connected to a single piece of medium. When one host transmits a signal onto the bus, all of the hosts on the bus can hear the signal; thus there is no need for routing. All but the intended hosts discard the message. The transmission spreads out from the transmitting host to the ends of the medium where it is electrically discarded. In principle, the bus may be any fully connected acyclic graph, rather than a linear connection topology.

The circular structure uses unidirectional communication paths; each interface must receive and then forward signals around the system. Messages need no routing since any message will pass its destination within one circuit of the system. Because the loop has no electrical terminus, some other mechanism must guarantee removal of messages from the medium.

The two members of the circular family are loops and rings. Although the distinction is often tenuous, a loop is a circular system with some form of central control; a ring is a system without any distinguished nodes. Part of the reason that distinguishing between the two is often difficult is that different loops have different levels of control in the control node. In some systems, the loop controller's function is supervisory or diagnostic; the controller only deals with anomalous conditions. In other systems, the loop controller is essential to the proper operation of the electrical signalling system used to transmit data around the loop.

Given these broad functional outlines, we can identify the following components in the communication system:

1. The medium - its topology, properties, and connection rules
2. Access control - the protocol for reading and writing messages onto the medium
3. The interface to the host - host control of the interface, the transfer of data between the interface and the host, presentation of status and events by the interface to the host
4. Buffering and formatting - rules that form the part of the message structure used in the communication system

Tables 2-1 and 2-2 present data relating to these four components. Discussion of the common aspects of these components is contained in the next five subsections following these tables. This discussion is followed by network-specific information. The common discussion and the table data relate to the four interface components as follows:

- *The medium.* Systems are divided by overall medium topology, with bus systems in Table 2-1 and circular systems in Table 2-2. Other properties of the medium are described by the "Medium Type & Speed" columns in the tables. Medium component selection is discussed in the "Media Characteristics" subsection. Techniques relating to medium use are discussed in "Signalling."

- *Access control.* The "Access Protocol" and "Control Method" columns furnish a brief description of the algorithm used to control transmission and the nature of any control elements. The abbreviation "dist" appears for any system in which control is totally distributed into the interfaces around the network in an essentially equal manner. General principles relating to access control are discussed in the "Access Control" subsection. Discussion of specific algorithms is delayed until the discussion of specific interfaces.
- *Host interface.* The method used by each interface to communicate data to its host is described in the "Host Interface" column. "RS-232" means a full-duplex serial connection according to the standard [RS-232C 69]; "byte" means some sort of byte wide parallel path; "DMA," "channel," and "microcode" indicate a parallel interface using the specified control method. Issues related to this data path and the necessary control paths are discussed in the "Host Interface" section.
- *Buffering and formatting.* In addition to simply passing bits between the host and the medium, the interface may buffer or process the data as it passes. The "Device Architecture" column specifies the main control element of the interface. The term "logic" means that the device is built of random logic, typically TTL, and hence is not easily changed, and "prog logic" denotes a custom controller that is programmed using PROMs or FPLAs. Several MOS microprocessors (e.g., 6800, 6502, Z80) and bit slice bipolar designs (e.g., 2900, 8X300) are represented in the tables. The "Buffer" column describes the amount of buffering found in the interface: "packet" means a single packet buffer; "micro memory" means that the available buffer space is equal to the size of microprocessor main memory less space used for programs, etc. The "Buffering and Formatting" subsection covers these considerations.
- The "Year of Ref" column gives the date in which the main reference for the system appears. Most of these systems have been implemented, but there are a few paper designs. In cases where conflicting information is available, the most optimistic set of consistent facts is presented.

Media Characteristics

The most frequently used media for local networks are twisted pair and coaxial cable. Fiber optics are generally regarded as promising, but are not yet widely used.

Twisted pair is the simplest medium, and is formed by twisting a pair of conductors. The twisting imparts a reasonable level of noise immunity to the cable; in critical applications shielded twisted pair is often used. Twisted pair is the easiest medium to work with, and can be easily spliced, bypassed, and terminated. Many buildings have spare twisted pair already pulled in cable trays. The cost of the line varies with the gauge of the conductor and whether shielding is required, but typically runs a few cents per foot. Special connection hardware is not required.

Coaxial cable is the basis for the majority of bus systems and several circular systems. It offers higher bandwidth and superior noise immunity than twisted pair. Because of its uniform impedance, it is vastly superior to twisted pair for multidrop systems. Coax is not as convenient to work with as twisted pair. Most coax systems are built using components developed for the Community Area TV

Table 2-1: Bus system features

Network Name	Medium Type & Speed (Mbps)		Access Protocol	Control Method	Device Archi-tecture	Host Interface	Buffer	Year of Ref
Ethernet	coax	3	contention	dist	logic+ micro-code	micro-code	none	1975
HYPER-channel	coax	50	reservation+ contention	dist	custom micro	channel	4-8KB	1975
ENET	coax	3-5	contention	dist	6800 micro	byte, RS-232	micro memory	1977
NBS network	coax	1	contention	dist	6800 micro	byte, RS-232	micro memory	1978
BATNET	coax	3	contention	dist	logic	DMA	none	1978
MITRE	dual coax	.3	contention	dist	6502 micro	RS-232 DMA	micro memory	1979
Chaosnet	coax	8	priority& contention	dist	logic		packet	1979
GMAD network	coax	<.1				RS-232		1979
Consortium Ethernet	coax	10	contention	dist	custom LSI			1980
Salplex	coax	<.2	contention	dist	custom LSI			1980

(CATV) industry. CATV connectors, taps, and cable are readily available and inexpensive due to mass production. One article [Anderson 79] estimates the cost of CATV cable at $.30/ft with a cost of $1 to $5 per tap. A MITRE technical publication [Holmgren 79] reports a cable cost of $.28/ft and a tap cost of $9.

Fiber optic media offer the ultimate in noise immunity and bandwidth. Systems have been built with data rates in the 100 Mbps range [Rawson 79, Okuda 78]. Because the medium is nonelectrical, fiber optics can be installed without reference to building codes, safely used in explosive environments, and inherently provide maximal electrical isolation between stations. This isolation avoids signal ground problems and concerns about lightning, static and other electrical hazards. It is only a drawback in those cases where it's desirable to power network devices with the same cable used for signalling, as is done in the systems used by the General Motors Corp. in its assembly plants [Smith 79].

Fiber optics technology is less mature than electrical technology. Standards are not well developed for any of the necessary components. Low-cost fiber optic line driver and receiver components have been developed, but these parts are aimed mostly at the terminal speed market, and their limited switching speed and output power constrain line speed and length. Laser transmitters have much greater capabilities, but at a correspondingly higher cost. Available components support point-to-point connections rather than the common bus connections, although

Table 2-2: Circular system features

Network Name	Medium Type & Speed (Mbps)		Access Protocol	Control Method	Device Archi-tecture	Host Interface	Buffer	Year of Ref
Newhall	twisted	3.1	token	monitor station				1969
Pierce	twisted (T1)	1.5	slotted	control station				1972
Spider	twisted (T1)	1.5	slotted+ message chop	control station	TEMPO mini	special	iface+ control	1972
DCS RI	twisted	2.2	token	dist	logic	DMA	none	1972
DLCN		1	variable msg insertion	dist	2900 slice		1 max message	1975
Cambridge loop	twisted	10	slotted message chop	monitor station	8X300 micro	DMA, byte	micro memory	1977
PRIMENET	coax	10	token	dist	logic	DMA	packet	1977
UCI&MIT LNI	twisted	1	token	dist	prog logic	DMA	64 byte FIFO	1977
Toshiba RCB	fiber optics	100	slotted	control station	logic			1978
NS LNI	twisted	2	token	dist	prog logic	DMA	64 byte FIFO	1978
TRW	fiber optics	20	slot insertion+ message chop	dist	logic	DMA	none	1979
IDA		40	slotted		logic	DMA RS-232		1979

[Rawson 79] speculates that this is not a fundamental restriction, and that networks with 25 to 50 passive taps are possible today.

The cost of fiber optics has been estimated to be on the order of $1/ft and $750 for the transmitter and receiver pair [Anderson 79]. A system capable of operation from DC to 10 Mbaud for up to 1000 meters is advertised by Hewlett Packard to cost $2/meter of cable, $385 per transmitter and receiver pair, plus a moderate additional cost for connectors, finishing, etc. [HP 1980]. This price is for quantities of 100, and hence shows only modest improvement since the Anderson estimate. As yet fiber optics are justified only in special circumstances, but will inevitably improve greatly in terms of ease of use, cost, and performance in the near term.

Signalling

The abstract function of the medium is to carry bit-serial data between the network interfaces of the network's hosts. Achieving this functionality requires a set of low-level conventions for transmitting and receiving the binary data. These conventions must deal with the nonideal properties of the medium, such as noise and delay, and convert the signal levels and conventions appropriate for the medium to the signal levels and conventions of the digital logic in the network interface. The main issues are the following:

1. Line termination
2. Electrical signal levels and isolation
3. Bit timing and synchronization
4. Modulation technique

Line termination

Although the length of the medium is short in comparison to that used in long haul networks, it is long in comparison to the rise time of the signals we wish to transmit over the line. Technically, such a medium is called a transmission line, and requires more careful treatment than that required of signals sent within a circuit board, processor, etc.

Signals on a transmission line are reflected by any nonuniformities in the medium and by the ends of the medium. The reflections are caused by the nonuniform impedance of the line at these points. If these reflections are large enough, they will interfere with the transmitted signal and make it incomprehensible. While transmission line reflections are always present to some extent, prudent design reduces them to an acceptable level.

Reflections from the ends of the medium can be controlled by terminating the medium with an appropriate resistance. The terminating resistance makes the medium appear to be electrically endless, and hence discards the arriving signal. In a point-to-point (unshared) transmission line, this termination is naturally combined with the line receiver on the end of the medium. In a multidrop transmission medium, completely passive terminators are attached to the ends of the medium if no receiver and terminator combinations are needed at these points.

Nonuniformities in the transmission line are caused by cable defects, cable connectors, taps, bypass relays, and switches, and anything else connected to the line. Careful component selection is necessary to ensure proper operation.

Termination effects mean that a failure of a medium component, even a passive one, can result in the complete failure of the medium. For example, if the central cable of a bus system is cut, the result will be two inoperable subnetworks, rather than two separately operable fragments. This result is inevitable unless more complicated communication medium transmission schemes are used. Another way to escape this problem is to greatly reduce the transmission speed of the medium. Because both of these "cures" are unpalatable, and the "disease" itself is rare, the usual course of action is simply to attempt to protect the medium from damage. If reliability is essential, redundancy of interfaces and links is the best approach

Electrical signal levels and isolation

If the network uses electrical links such as coax or twisted pair, the electrical properties of the medium must be considered. The problems are caused by the fact that each station on the network has its own local ground reference. Ground differences of up to 10 volts are common, and differences in the 50 volt range are found in rare cases.

In general, the difference is correlated to the length of the communication path. Systems on the same power circuit will have virtually no differential, systems in the same building will have a smaller differential than those in different buildings, etc. Large electrical machinery and other heavy loads on the power distribution system can cause short-term variances in ground potentials.

These differentials are the source of two undesirable effects: The signal on the communication line is more difficult to recover, and a current corresponding to the ground difference is induced on the line.

The signal is more difficult to recover because the different voltage levels that correspond to ones and zeros must be measured with respect to some signal ground. The uncertainty in regard to the ground potential at the line receiver versus the ground potential at the line driver can get to be larger than the difference between the zero and one voltage levels at the line driver. One solution is to simply increase the size of the margin between a one and a zero so that the reference difference is made insignificant. While this approach works, it has numerous drawbacks: It doesn't eliminate the spurious current flow, it may require extra supply voltages, and it requires that different logic levels than those in the main interface hardware are used.

A refinement of the first approach is to use differential signalling. In this method, the signal is defined to be the difference in potential between the two conductors that make up the communication link. This eliminates common mode noise, i.e., anything that changes the potential of both conductors equally. Differential line drivers and receivers are readily available; hence this is a low-cost scheme that will handle differences of 10 volts or so. The RS-232 signalling protocol is an example of such a system. The drawbacks of this approach are that it will usually require another supply voltage and that it delivers a worse combination of data rate, distance, and noise immunity than other, more sophisticated systems.

Another approach is to couple to the medium with a component that eliminates the DC path between the medium and the interface. One such component is the opto-isolator, which consists of a coupled LED and photo-transistor. While this approach offers several thousand volts of isolation, there are drawbacks. The LED draws significant power from the line and presents a nonlinear impedance that makes impedance matching difficult without an active compensation network. If an active compensation network is used, it must be powered and hence creates a new isolation problem. Currently available opto-isolators are limited to a few Mhz of bandwidth. Transformers pose similar problems.

One way to sidestep the ground reference problem is to build a section of the interface that is referenced to the medium ground and not directly connected to the local ground of the host. This section is powered using its own floating power supply. Although the signal must still be coupled from this section to the rest of the interface, this is a simpler problem because the distance is small; hence transmission line effects can be ignored.

This is the approach adopted in the majority of systems. The separate section, called the transceiver, solves the isolation and signal levels problems. The transceiver is also a convenient module in terms of the abstract, and often the physical, organization of the interface.

Bit timing and synchronization

In addition to correctly interpreting the incoming signal levels, the interface must be able to determine the boundaries of the incoming bit cells.

One possibility is for the receiver and transmitter to have synchronized clocks for determining the bit cell boundaries. The synchronization can be preserved by distributing a centrally generated clock, or by periodically correcting drift among individual clocks. Increasing the propagation delay or the data rate increases the difficulty of maintaining synchronization.

The alternative is to send a clock signal along with the data.

The clock can be sent a completely separate signal on a separate cable. This approach doubles the medium cost, and requires the consideration of possible skew problems.

A more elegant solution is to combine the clock with the data in such a way that the two can be easily separated at the receiver. Several schemes exist for doing this. One of these systems, a member of the Manchester code family, is illustrated in Figure 2-1. The transmitting interface constructs the signal to be placed on the medium by EXclusive ORing the data and a clock at the transmitter.

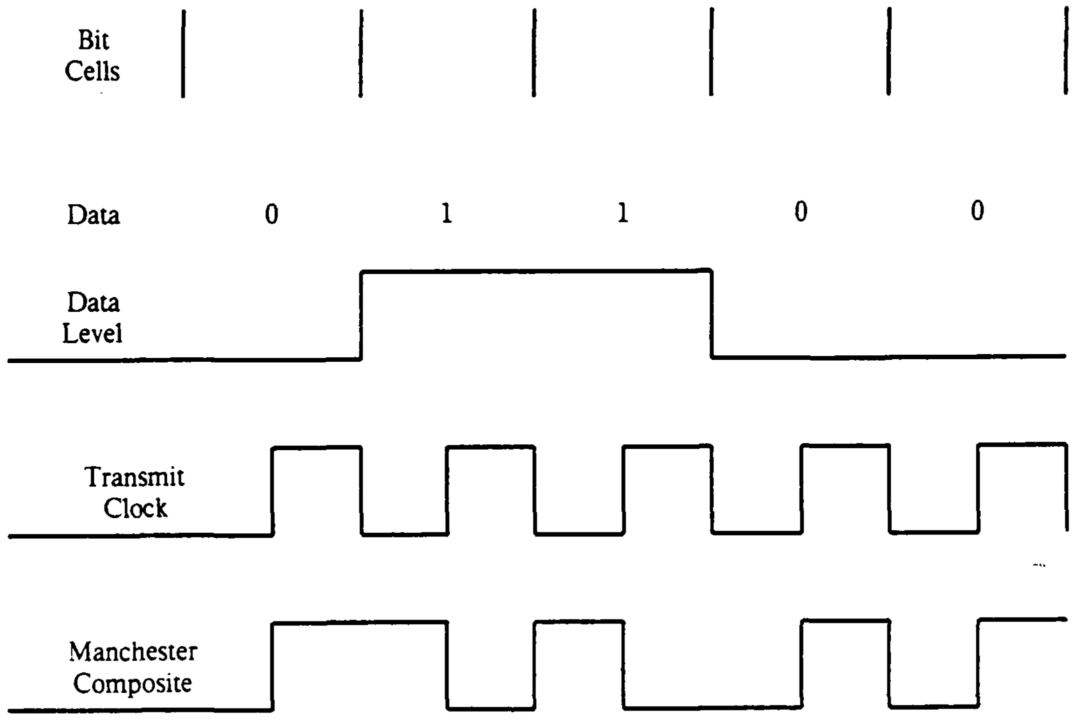


Figure 2-1: Manchester coding

The transmit clock has a transition in the middle of the bit cell; the transmitted data does not. Thus the composite signal will always have a transition in the middle of a bit cell. Depending on the data being transmitted, there may be a transition at the boundary between bit cells.

Once the receiver has found a mid-bit transition, it tries to find the next mid-bit transition. To do so it must avoid the possible transition between bit cells. The usual way to do this is to ignore transitions that take place too soon after a mid-bit transition. For example the receiver might wait 75 percent of a bit time after a known mid-bit transition before looking for a new mid-bit transition. The arrival of the new transition is used to restart the 75 percent timer for the next search, etc. The timeout interval insures that data transitions are not mistaken for mid-bit transitions. The timing of this interval is not very critical and clock drift is not a problem since each mid-bit transmission resynchronizes the system.

This leaves the problem of finding the first mid-bit transition. One way to do this is to require that all transmissions be prefaced with a known pattern. For example, using the code of Figure 2-1, if all transmissions start with a zero, then the first transition in each transmission will be a mid-bit transition.

The only drawback of these self-clocked transmission systems is that the amount of bandwidth required to carry the signal is doubled. This usually isn't a problem because of the high potential bandwidths, so most local networks use self-clocked transmission systems. Systems that send a separate clock usually do so because their data rate is very high (hence the doubled bandwidth requirement is too large for a single medium) or because the distance being covered is short enough that clock skew and the cost of the additional medium are negligible.

Modulation technique

The composite clock and data signal produced by the Manchester code represents all of the digital information the interfaces on the network wish to exchange. If this signal is directly coupled to the medium, the signal is said to be transmitted using baseband modulation.

The alternative is to use the Manchester composite to modulate a carrier frequency. Carriers are usually chosen to be compatible with the TV frequencies used in CATV systems so as to take advantage of the availability of CATV hardware.

The most common design uses a cable topology that resembles a tree; network interfaces transmit their signals toward the root (or head-end, in CATV jargon) of the tree, which repeats the transmitted signals back toward the leaves of the tree. Two sets of frequencies are used: one for transmission toward the root and one for transmissions from the head-end to the nodes. The separate frequency bands allow the use of simple analog splitters at the nodes of the tree. If channels use standard 6 MHz TV channels, about 15-20 channels can be created, each with a separate allocation for up and down channels. Assuming typical signal-to-noise ratios, the 6 MHz channels can carry up to 10 Mbps, although less efficient use is much cheaper. For this reason, typical systems use a 6 MHz channel for 100-300 Kbps data rates.

For example, the General Motors Corporation Assembly Division (GMAD) system described in [Smith 79] uses frequencies in the 11-120 MHz and 160-300 MHz parts of the spectrum, while carrying data rates in the 100 Kbps range.

The advantage of the baseband modulation is that it doesn't require any receiving and transmitting RF modems, and hence may be the simplest and lowest cost system. The advantages of carrier systems are derived from the well-defined frequency of the carrier:

- Carrier-based signals have a more uniform power spectrum with respect to frequency and hence are propagated along the cable in a more uniform manner. This means that longer distances for a given data rate are possible.

- Multiple carriers and signals can be carried on a single cable using frequency division multiplexing (FDM). One advantage of FDM is that the multiple channels it creates need not interface or even be aware of each other. The channels need not even use compatible signals or equal bandwidth. For example, one channel can be carrying a standard video signal whilst another channel is carrying low-speed terminal traffic. The local network can use FDM to create independent channels if this is architecturally desirable. A common use for this technique is to separate transmissions going to and coming from a central controller.
- The high frequency of the carrier-based signal eliminates the need for any DC signal coupling to the medium and may help to solve the isolation problem.
- The builder of a local network may be able to share an existing cable system so as to avoid the cost of laying cable.

In most cases these advantages either aren't of interest or aren't cost effective; hence baseband systems predominate. Carrier systems are used where sharing of the medium through FDM is required. In any case, the problems of building the local network are essentially identical whether one is using a channel in a carrier system or the baseband on a dedicated cable.

Access Control

Because the medium's bandwidth is a shared resource, access control is necessary to insure that the interfaces use the medium in an orderly and efficient manner. The realization of this mechanism may be interface hardware, interface software, host software, a special device, or some combination of these. From the microscopic view of the interface, the access control mechanism grants the right to transmit some time after the interface requests that right. From the macroscopic view of the network, the job of the access control mechanism is to allocate the medium's bandwidth in a manner that delivers maximum utility to the system.

In long-haul networks, this maximization involves complicated tradeoffs between throughput, delay, and fair allocation to all users on the network, and is complicated by the interactions between store-and-forward nodes, multiple routes, buffering, etc. In a local network the problem is simpler; access control must be able to arbitrate rapidly between multiple interfaces requesting service, and should grant access fairly. The only part of the problem that is harder in a local environment is that the higher medium speed creates the need for correspondingly fast access control.

The implementation of access control is the single largest source of diversity between local networks. Among the causes of this diversity are different choices as to the allocation units, the allocation strategy, and the distribution of the access control function.

The allocation unit is usually either a fixed-size slot or a message or frame whose size corresponds to the amount of data the sender needs to transmit. Fixed-size slots are usually found in systems with a central allocator which is responsible for generating the slot boundaries. Small fixed-size slots can implement a TDM analog of multiple FDM channels. The advantage of the fixed-size slot system is that it requires few actions by the interface, and hence few components in each interface. The small amount of control can be important in very high-speed systems where it is imperative to minimize the number of very high-speed components because of their higher cost. Although other parts of the interface can be built using parallel logic, the medium interface and its access control are inherently serial for a serial medium, and hence simplicity is the best way to reduce the cost of such a system.

The allocation strategy can be characterized as static or demand based. In static allocation, some portion (e.g., FDM channel, every third slot, every fifth message) is allocated to a particular use. In demand allocation, bandwidth is not allocated until it is needed or requested. These choices represent points on a spectrum rather than a binary decision; many schemes offer elements of both strategies or use different strategies in different situations. An example of the first combination is a system which allocates fixed bandwidth for the duration of a conversation or file transfer. The latter combination has been suggested for use in supporting real-time traffic such as speech or video on the same communication medium used for conventional data transfer.

The allocation strategy can be characterized as being centralized or distributed. Centralization has inherent advantages: Decisions are made in one place and hence are always consistent and coordinated. Although centralized access control represents a cost that isn't replicated in each interface, it also represents a single point of failure. Other disadvantages include the need to distribute decisions and to collect information for enlightened decision making. These needs imply the existence of separate communication channels for these functions and may present a throughput problem.

A decentralized access control system solves some of these problems by its very nature, but it also creates some new concerns:

- The distributed algorithm won't have superior reliability unless it can compensate for failures in individual interfaces. The design should be modular and self organizing, so that removing any interface or adding an interface to the network doesn't stop operation of the access control mechanism.
- The multiple decision locations will be making decisions on the basis of different information. The access control mechanism must be robust enough to cope with the inevitable conflicts that occur.
- It is more difficult to develop, debug, and measure the performance of distributed algorithms. Thus it can be much more difficult to have the same level of confidence in a distributed access control's ability to operate correctly in the face of error, varying load conditions, and real-time constraints.

Host Interface

The host interface is the actual connection between the network interface and the host. The design of the host interface involves consideration of the data to be passed and the available interface choices.

The nature of the communication task constrains the nature of the connection between the host and the interface. In general we can assume that there exists independent communication flow into and out of the interface and that it may be necessary to allow equivalent traffic into and out of the host. The input and output communication paths are essentially symmetrical; whether they are implemented as a full-duplex path or a shared simplex path is of little consequence. In any case, each side of the path carries four types of information:

1. *Control.* Information from the host to the interface that controls the operation of the interface. This information includes commands to start data transfer, reset the interface, set the interface address, etc. The amount of information of this type will typically be a few bytes per message transfer, hence its transfer rate is of little importance.

2. *Interrupt.* If the host is to be able to process other work while the interface operates, the interface needs to be able to interrupt the host when conditions specified by the host in the control information are met. The methods for generating interrupts are host specific, as are the data that needs to be transferred. The possible data elements include the interface's I/O address, interrupt vectors, priority codes, and status information.
3. *Status.* This is information to the host that tells the host the status of requests. A few bytes of this information are transferred per message. Certain changes in status must be coordinated with interrupts.
4. *Data.* This path carries the actual message data. Because it carries the largest amount of data, its rate is the most important.

The methods available for transferring this information vary from host to host. The designer will often use more than one of the available methods to complete the host interface. The available methods include the following:

- *Shared memory.* This method is applicable for hosts which include I/O devices in the address space of the memory. Hosts for which this is true include some minicomputers, such as the PDP-11 [DEC 79] and Lockheed SUE [Lockheed 72], and almost all of the MOS microprocessors. The type of path looks to the host to be one or more special memory locations, and looks to the interface as a set of one or more registers or register files. This method is often used for control and status information. When used for data, the message can either appear as a large memory region or as a register which automatically advances to the next message byte or word when accessed by the host. If the latter method is used, the timing of transfers must be considered to avoid possible loss of data due to slow response by the host or the interface. The implementation of this approach is simple for those machines where it is appropriate, and implementations for one host can be easily modified to fit another host if the host memory units are similar.
- *Direct Memory Access (DMA.)* In this method, the interface is able to access the host's memory directly without interrupting the execution of instructions by the host. The usual method for doing this requires the interface to request access to a specified memory location via the host's I/O bus, wait for a go-ahead signal from the I/O bus, and then follow a bus protocol to read or write the specified memory location. Implementing DMA usually requires adherence to a fairly complex and time-sensitive protocol, and hence a fairly high level of intelligence. Hardware implementations of DMA require at least several LSI chips (if the appropriate parts are available), or on the order of 50-100 MSI parts. Failure to adhere to the appropriate I/O bus protocols can hang the host or cause incorrect execution. In summary, DMA provides the highest data rate possible without the use of dedicated memory locations, but at the highest cost in terms of parts count and complexity. DMA designs are not transferable between different host types, and are sometimes not transferable between different models of the same host.
- *RS-232.* RS-232 is the standard protocol for interfacing terminals to hosts. As such, it has the advantage that an RS-232 interface is almost always transferable to any host. In addition, LSI components for implementing this protocol are readily available and low in cost. The data rate of the RS-232 protocol is limited; for most hosts 19.2 Kbaud is the highest possible rate to run RS-232. The protocol is suited to the transfer of sequential characters; many host systems restrict the set of characters that can be passed to those having the correct parity, or to those characters that are printable. Interrupts can be implemented through the use of the break function; this is supported by most, but not all,

hosts. This choice is in general only appropriate for interfacing terminals or other very low-speed "hosts."

- *Channels and special I/O busses.* Many large hosts and some minicomputers implement I/O functions with a usually unique bus protocol. Although some *de facto* standards exist, such as the IBM channel protocol and the S-100 bus, these systems usually are similar in nature but different in terms of signal levels, timing, number of signal lines, etc. Throughput varies depending on design. This is the only method for generating interrupts on most computers.

Buffering and Formatting

In addition to providing a path for data transfer between the host and the medium, the interface may also be required to do some buffering or processing of the data.

A small amount of buffering is usually necessary just to convert between the bit serial medium format and a byte or word parallel data format used by the host. Buffering is also necessary to compensate for any differences in data rate between the medium and the host. Given that the medium's data rate is essentially constant, the difference has two components: transient variations and steady state variations.

Transient differences manifest themselves when the interface requests a byte or word transfer from the host. Regardless of the host interface type, there will be some nonuniform response time from the host, because the host may be asynchronously processing instructions and other I/O requests, and at the minimum requires a variable amount of time to synchronize the request with the host's internal data paths. For example, variations in interrupt response time can be several milliseconds, and even DMA transfers can require several microseconds. These times can be reduced, but not eliminated, by careful selection of DMA or interrupt priorities, special scheduling of network I/O, etc. Thus, at megabit rates, there will be at least several bit times of delay between presentation of data at the host and its consumption. The usual practice is to solve this problem with at least a host memory unit's worth of buffering at the periphery of the serial-to-parallel converter.

Steady state variations in data rates stem from higher level design issues. The systems architect has to balance three rates:

1. *The medium data rate.* This has an absolute upper bound based on the choice of medium and medium components such as line drivers and receivers, and design constraints such as distance.
2. *The processing rate of the interface.* This rate is dependent on the cycle time and width of the interface's internal data paths, and the amount of processing performed on the data. It varies from a few Kbps for a serial data path managed by a MOS microprocessor to hundreds of Mbps for custom high-speed logic.
3. *The host data rate.* The rate is bounded above, and possibly below, by the transfer rate of the host interface.

Any differences in these rates result in a need to buffer data in the interface. Even in the rare cases where it is possible to select a single data rate for all three paths, there are often reasons for not doing so:

- Buffering may be cheaper than speeding up the bottleneck. For example, MOS microprocessor implementations have essentially zero cost access to RAM memory.
- In order to satisfy the bandwidth requirements of all hosts on the network, the bandwidth of the network may need to be higher than the highest possible host interface bandwidth.
- Different hosts may require different levels of service. For example, a terminal might not need the high throughput needed by a large server. This leads to the possibility that interfaces might not be homogeneous throughout a network.
- If the interface filters out messages that aren't addressed to the attached host, it must at least be able to buffer the address portion of the message.
- Interfaces may need to buffer some or all of the message in order to process it, provide for automatic retransmission, etc.

Thus, consideration of the buffering problem leads to the question of how much processing the interface is to do on the data being transferred. In some networks, the interface doesn't even understand the concept of message boundaries. This leads to the least possible amount of work in the interface section, and passes on problems to the host. In other systems, particularly microprocessor-based systems, the interface deals with one or more protocol levels for the host. Thus this issue is intertwined with the protocol hierarchy. The "average" level of interface intelligence is the message format or frame level. At these levels, the interface will see the following message format items:

- *The preamble.* A known pattern to bit synchronize the transceiver, and possibly to synchronize the receiving interface to the start of the message.
- *The destination address.* A field that specifies which interfaces are to receive the message.
- *The origin address.* The "signature" on the message which denotes its source. This field is usually sent in the same format as the destination address.
- *The data length.* An optional field that may be necessary if the data field can have a variable length.
- *The data field.* The possibly variable amount of data.
- *CRC or checksum.* Included to allow the interface to recognize damaged transmissions.
- *The postamble.* As required by the transmission system.

2.2 BUS SYSTEMS

Introduction

The source of many of the ideas found in today's bus networks is the ALOHA network at the University of Hawaii [Abramson 70, Abramson 73b]. The ALOHA network started as an experiment to test the use of packet radio as a means of connecting terminals scattered around the Hawaiian islands to a front end computer, the Menehune, and hence to a central computer. The ALOHA system influenced later local network designs in its use of the radio medium.

In the ALOHA network two independent channels share the radio medium: a radio channel that the

Menehune uses to talk to all of the terminals, and a separate channel that the terminals share and use to talk to the Menehune. The channel from the Menehune is easy to manage; messages are transmitted one after the other with queueing in the Menehune when necessary. The shared channel to the Menehune is more difficult to manage because of the multiple sources of transmission.

In selecting an access control system for the shared channel to the Menehune, the ALOHA designers decided to avoid systems that explicitly allocate the channel, such as Time Division Multiple Access (TDMA), which allocates time slots to each source, and polling, which has the control node periodically "ask" each node for a transmission. Instead, terminals statistically contend for the medium. In the "pure ALOHA" access control system, a terminal with a message to transmit simply turns on its transmitter and transmits. The terminal then waits for some interval for an acknowledgment. If an acknowledgment returns within the interval, the message has been successfully transmitted. If not, the terminal retransmits the message and waits again. The retransmission and wait cycle is repeated until an acknowledgment arrives.

The eventual success of this scheme is guaranteed by the relentless retransmissions so long as each retransmission has some chance of success. Random errors, such as those caused by radio noise, are corrected. Errors caused by multiple overlapping transmissions are also cured, so long as the system avoids an infinite sequence of synchronized retransmission collisions. In pure ALOHA, this is avoided by having the terminals use different timeouts; the macroscopic access control system actually relies on there being differences between the microscopic access control mechanisms.

The main drawback of the pure ALOHA system is its low channel utilization; the collisions between sources in a saturated system lead to a maximum utilization of 1/2e, or approximately 18 percent. This utilization can be doubled to 1/e, or 36 percent through the use of "slotted ALOHA." In slotted ALOHA, sources cannot transmit at any time; instead, they can only begin transmitting at the start of discrete time slots. Slotted ALOHA has a superior performance because a given transmission can only collide with others during one slot, instead of overlapping and interfering with two transmissions.

Overview of the Ethernet

The ALOHA experience suggested a local network structure to several researchers at Xerox. Why not use a piece of coaxial cable as a private radio universe?

The result of this idea is the Ethernet [Metcalfe 76]. The Ethernet is probably the best known, and most copied, form of local network in use today. It is used at several sites within Xerox, as well as MIT, Caltech, and several other universities. The Ethernet is used as a basic building block in an internetworking system that spans "about 1000 computers, on 25 networks of 5 different types, using 20 internetwork gateways" [Boggs 79]. Another measure of the success of the Ethernet is the large number of host types that have been interfaced to it, including Xerox's ALTO minicomputers [Thacker 79], PDP-10-size computers, printing servers, file servers [Swinehart 79], and other machines.

The most well-known version of the interface is that used to connect the ALTO minicomputer to the Ether. This version is described in the following sections. A block diagram of a small Ethernet system is shown in Figure 2-2.

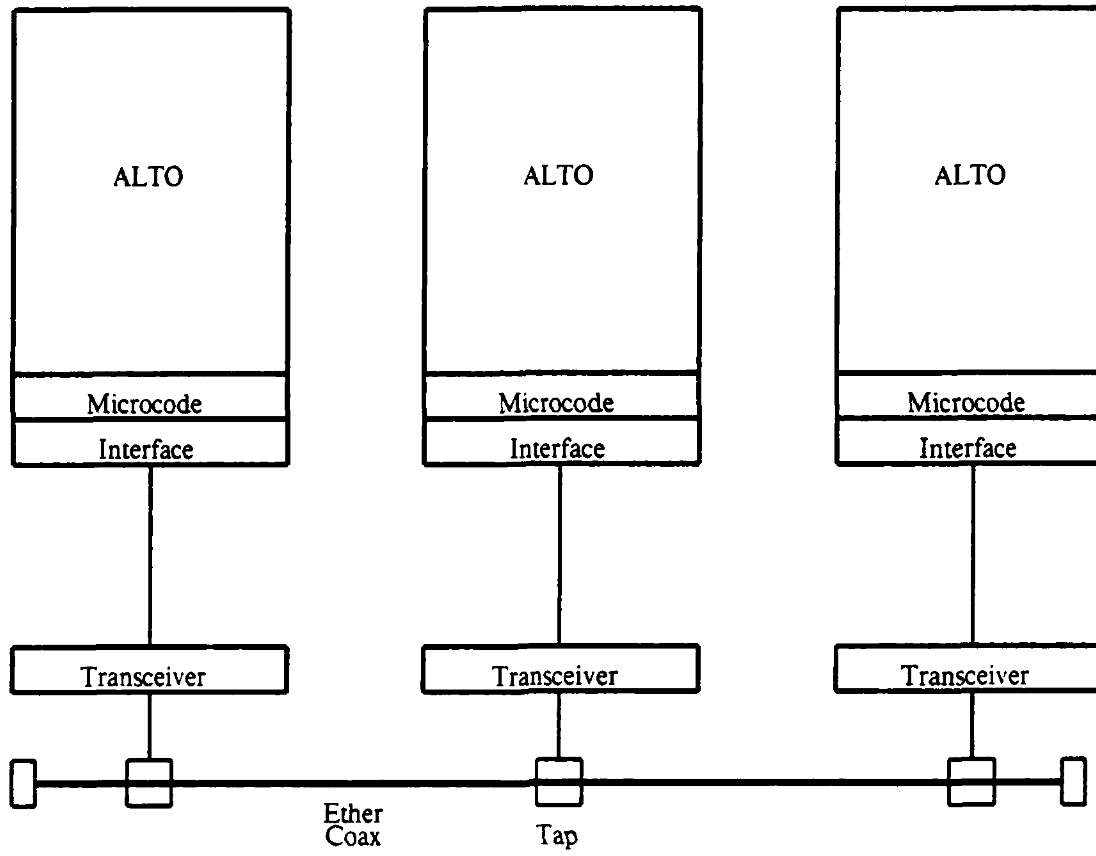


Figure 2-2: Ethernet components

Ethernet Medium

The communication medium of the Ethernet is a piece of coaxial cable. The design goals for the system include the ability to connect 100 hosts at 3 Mbps along a kilometer of Ether [Metcalfe 76]. As currently implemented, the Ether uses RG/11 75 ohm foam cable and Jerrold CATV tapping hardware [Crane 80]. The interface is connected to the medium through a separate transceiver assembly. The transceiver is powered by a floating power supply which is referenced to the cable ground. The transceiver communicates to the interface over twisted pair lines that are decoupled using transformers that eliminate the DC path.

The topology of the Ether is constrained to be an acyclic graph by the need to avoid multipath interference. In practice, the Ether is further constrained to be linear, thus avoiding power loss and reflection problems at the branch points.

Data on the Ether is transmitted using a Manchester code. In order to synchronize the receivers, a one-bit synchronization preamble is sent before every message.

Ethernet Access Control

Although derived from the ALOHA system, Ethernet operation is different in several important ways. The first difference is the equality of hosts. There is no distinguished node similar to the Menehune; all hosts contend for the single communication channel. Any host on the Ethernet can communicate with any other. The second difference is that the Ether has different properties than the radio system used by the ALOHA network:

- The noise level of the Ether is much lower; there is less random noise and no crosstalk from other users of the medium.
- Any Ethernet interface can hear all of the transmissions on the Ether; the UHF radio used by ALOHA needs a clear line of sight for transmissions. Islands cut the line of sight between some ALOHA terminals.
- The propagation time of signals on the Ether is much lower.

These properties allow for new access control mechanisms that make more efficient use of the medium than is possible in ALOHA. The efficiency of the Ethernet system is very dependent on message size and medium length and asymptotically approaches the 37 percent ALOHA behavior for very small messages. However, empirical measurements [Shoch 79] have shown that the Ethernet system delivers 56 percent efficiency for the smallest sized messages actually used in the Ethernet environment (6 bytes), and 97 percent efficiency for the largest size messages (approximately 570 bytes).

The access control mechanisms of the Ethernet are as follows:

1. *Carrier Sense.* When an Ethernet interface has a packet to transmit, it listens to the Ether to determine if the Ether is presently in use. If so, the interface defers to the transmission in progress until the Ether is silent. This avoids collisions that would serve no useful purpose. This access control strategy is called Carrier Sense Multiple Access (CSMA).
2. *Collision Detect.* As an Ethernet interface transmits its message, it listens to the Ether to determine if the signal on the Ether matches the transmission in progress. If not, the probable reason is that two or more interfaces have chosen the same instant to transmit and have thus caused a collision on the Ether. Because the transmission has already been ruined, it is pointless to continue, and the Ethernet interface terminates the transmission using the "Collision Consensus Enforcement" policy and reschedules the transmission as described under "Binary Exponential Backoff." Access control algorithms incorporating CSMA and Collision Detection are termed CSMA/CD.
3. *Collision Consensus Enforcement.* When a collision occurs on the Ether, the most likely assumption is that all transmissions in progress are ruined. However, it may be that the collision is only immediately apparent to one of the transmitting interfaces due to data dependencies. In order to ensure that all transmitting interfaces recognize the collision, interfaces that recognize collisions "jam" the Ether for a period of 35 microseconds following collision detection. The jamming signal is a continuous low voltage that violates the ordinary Manchester signalling protocol, but is likely to be seen by all transmitting interfaces. Following the jamming signal, the interface terminates its transmission.
4. *Binary Exponential Backoff.* After a collision, the interfaces which caused the collision wait for a variable amount of time before attempting a retransmission. The timeout should have two properties: (1) it should be different in different interfaces to avoid an

infinite series of lockstep collisions, and (2) it should increase as demand for the Ether increases, so as to restrict offered load as the medium becomes saturated.

The Ethernet interfaces calculate the interval to wait using an unsynchronized counter that is incremented at a high rate. When the interface needs a new timeout, it masks the value of this counter with a bit mask and stores the result in a timing register. The mask has only its low bit on for the first collision; every consecutive collision turns on one more bit in the mask, until the mask reaches a maximum value of eight ones. Once loaded, the timing register is counted down to zero before retransmission is attempted. The interval between counts is intended to approximate the round trip delay of the Ether. This procedure has the effect of selecting a random interval whose mean is doubled every time a collision occurs.

5. *Truncated Packet Filtering.* A side effect of the implementation of the previous strategies is that short bursts of data are found on the Ether that are too short to be valid, given a knowledge of the higher level protocols. Such "runt packets" are automatically filtered out by the interface.
6. *Packet Error Detection.* The transmitting Ethernet interface computes a hardware CRC that is included at the end of every transmission. Receiving Ethernet interfaces recompute the CRC and discard packets with a bad CRC.

Ethernet Host Interface

As previously mentioned, the Ethernet interface is a unique combination of hardware and microcode. As such, the interface has different properties depending on the level from which it is viewed. From the standpoint of the ALTO, the Ethernet interface is similar to a DMA interface. The Ethernet interface for the PDP-11 is a DMA interface for data and uses memory locations for status and commands.

The ALTO interface shares logic for input and output, and hence is really a half-duplex device. The ALTO microcode manages the sense of the interface; normally the interface is set to receive from the medium. When the host wants to transmit, the device is switched to transmit mode for the duration of the transmission. Packets addressed to an interface aren't received while the interface is in transmit mode. Crane and Taft [Crane 80] mention this as being a problem that becomes noticeable for servers and other hosts that run multiple concurrent communication paths. The Ethernet attempts to reduce this problem by special programming of higher level protocols. The same authors also mention that the interface is unable to receive a second packet immediately following a previous packet addressed to the interface; the problem here is that the microcode requires several milliseconds to re-enable the receive interface.

The half-duplex part of the problem stems from the desire to reduce the component count for the interface. New designs, built with newer (and denser) logic can avoid it at low cost. The problem of consecutive packets addressed to the same interface is a problem that is common to almost all local network systems; it is a tradeoff between interface complexity and performance that is very sensitive to the usage pattern of the network and the architecture of the interface.

Ethernet Buffering and Formatting

The ALTO Ethernet controller includes a FIFO with 16 entries of one word (16 bits) each, and a 16-bit register for serialization and deserialization of the data stream. The FIFO compensates for variances in the response time of the microcode.

The message format known by the interface and microcode is shown in Figure 2-3.

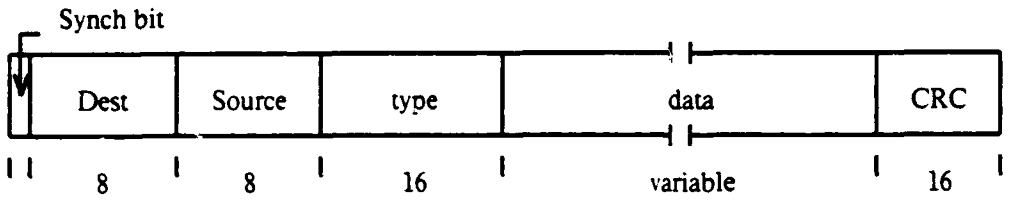


Figure 2-3: Ethernet message format

The synch bit is used to synchronize the receiver and isn't seen by the host. The destination and source addresses (Dest and Source in Figure 2-3) are 8 bits each. Each interface has its own 8-bit address that is unique throughout a particular Ethernet. In addition to receiving packets addressed to this address, the interface copies packets addressed to a broadcast address (all zeros). Thus a given message is addressed either to a specific interface, or to all interfaces on the Ethernet.

The Ethernet interface has a special "promiscuous" mode, in which the address filtering mechanism is disabled and the interface copies packets regardless of address. The promiscuous mode is mainly used for measurements of Ethernet activity.

The type field is used by higher level protocols and is essentially ignored by the interface. Its presence at this level is mainly for standardization purposes. The data field of the message is composed of a variable number of bytes. The interface finds the end of the message by detecting the end of the Manchester coding transitions. The CRC is appended to the message by the transmitting interface and is removed by the receiving interface before the message is copied into the host.

Other Bus Systems

BATNET

The BATNET is a network similar to the Ethernet. The BATNET has been implemented, and is in use, at the Battelle-Northwest Laboratories [Gerhardstein 78]. Using approximately 2500 feet of RG11/U coaxial cable, the network connects PDP-11 systems and runs at 3 Mbps. The cost of the interface components has been estimated to be approximately $1500, excluding labor costs [Gerhardstein 78].

The main components of the BATNET interface are a transceiver, minimal control logic, and a DEC PDP-11 DMA interface board. The interface is totally ignorant of message format and the access control system; the three operations possible with the BATNET interface are (1) setting an interrupt generating timer, (2) the output of a block of memory to the medium, and (3) the reception into

PDP-11 memory of the next block of data to appear on the medium. Address recognition and contention resolution are handled by interrupt routines in the PDP-11.

This design results in an interface which is minimal in terms of hardware component count at the expense of degraded performance and host overhead.

ENET and NBS network

Although developed independently, the ENET interface [West 77] and the NBS network interface [Carpenter 78] are quite similar; both are microprocessor designs that use a contention bus and access protocols similar to those used in the Ethernet.

The ENET (Expensive NET) interface is one of two network interfaces designed at the Queen Mary College at the University of London, the other being the CNET (Cheap NET). The ENET is suitable for computer networking, whereas the CNET is really a terminal multiplexor built around a contention bus. The ENET is only comparatively expensive; a cost estimate for the interface is 200 pounds, or approximately $450 [West 77].

The NBS network interface, developed at the National Bureau of Standards in Gaithersburg, is designed to connect hosts and terminals to a contention bus. It has two types of host interface: a RS-232 port for terminals, and a byte parallel port for hosts.

Both of these interfaces use a MOS microprocessor as the central control element of the interface. This strategy has several potential advantages:

- A dedicated microprocessor can replace much of the logic, and hence the cost, of the interface.
- Since the programming of the microprocessor determines the characteristics of the interface, the interface could be easily changed in the future by reprogramming.
- The microprocessor can implement higher order protocols in the interface than is possible in other types of designs.

Given these goals and the constraints imposed by the microprocessor, the interface consists of three parts:

1. *The dedicated microprocessor controller*, including program in EPROM, RAM for buffer space and program variables, and the requisite microprocessor support circuitry.
2. *Custom circuitry to allow the microprocessor to communicate with the host.* This section of the interface looks like a peripheral device to both the microprocessor and the host.

   Simple byte parallel and RS-232 interfaces can be built using standard LSI microprocessor parts at the cost of one or two LSI parts and a few SSI parts for address decode, bus drive, etc. This is the approach used in the ENET.

   The NBS design allows the dedicated control microprocessor to serve up to eight separate host interfaces. The individual interfaces are also microprocessor based, and two types have been built: one for computers and one for terminals. The computer interface is byte parallel and depends on the attached host for some protocol functions; the terminal interface uses RS-232 and includes all protocol functions necessary to allow the terminal user to access hosts on the network.

3. *Logic interfacing the microprocessor to the medium.* Most design decisions relating to this part of the interface are insensitive to the use of a microprocessor controller. For example, facilities for line driving, bit synchronization, collision detection, and carrier sense are needed and are implemented in a similar manner whether or not a microprocessor is used.

   The use of a microprocessor controller adds constraints on the design of the data path between the medium and the controller. This path provides for the transfer of message data bytes and for any residual translation between the serial data format on the medium and whatever form is used by the controller.

   The desire to reduce component count, and hence cost, suggests that microprocessor memory be used as much as possible for message buffering, and that the microprocessor should perform as much as possible of the formatting task. This aim often conflicts with the throughput requirements of the network. Even if the microprocessor can handle all of the local host's traffic, a medium data rate high enough to satisfy the transfer needs of all network hosts will usually be higher than the instantaneous rate that can be provided by the microprocessor.

   The most convenient data transfer method would be for the microprocessor to provide bytes to the medium interface under the control of interrupt driven I/O. This would allow the interface to multiplex medium service with the other activities of the interface in a natural way. However, this approach doesn't deliver an acceptable data rate. In the case of the 6800 microprocessor used in the ENET and NBS systems, interrupt context switches take approximately 20 microseconds, and the 4 or 5 instructions to prepare the next data byte take on the order of 15 microseconds. Assuming byte transfers, and a maximum latency of 2 interrupt context switches per transfer, the worst case time to respond to an interrupt is 55 microseconds. This corresponds to a data rate of less than 200 Kbps.

   This data rate can be improved by using the tightest possible program loop and ignoring all other activities for the duration of the medium transfer. Even with this strategy, the data rate remains under 1 Mbps.

   The only choice seems to be to remove the microprocessor from the data path to the medium, thus decoupling the data rate of the microprocessor from the data rate of the medium. This implies some form of shared memory or buffering between the microprocessor and the medium driver.

   In both the ENET and NBS interfaces, this sharing is implemented in the form of special sections of microprocessor memory and a multiplexed address bus. The special sections of memory run at twice the rate required by the microprocessor; alternate cycles are allocated to the medium driver. In essence, this is a special form of DMA which avoids the arbitration overhead of more general DMA schemes, and can be implemented with few components.

   The NBS system uses a Signetics 2651 SDLC chip for the data path formatting functions. Although the frame format isn't ideal, the one chip cost is attractive. The ENET uses custom logic for serialization and deserialization.

Chaosnet

The Chaosnet is a network built at the MIT Artificial Intelligence Lab to connect the lab's PDP-10, PDP-11, and LISP machines [Shoch 79]. The network spans a kilometer of coax and runs at 8 Mbps.

The higher data rate and added length makes the medium less robust than the Ether; tuning is required when a new host is added to the network.

The 8 Mbps data rate is also faster than the maximum possible DMA rate for some of these machines; hence full packet buffers are incorporated in every interface.

The access control system of the Chaosnet is a hybrid between the Ethernet contention system and a TDMA approach. While the interfaces use carrier sense and collision detection, they also attempt to sequence access to the medium in order to avoid collisions.

Each interface has a unique ID number and a counter that sequences through the range of allocated ID numbers. The counter's value corresponds to the ID of the interface that "owns" the medium. The counter advances at a high rate as long as the medium is free. When an interface wishes to transmit, it waits for its counter to equal its ID and for the medium to be free. A successful transmission loads all counters with the ID of the source of the transmission.

Collisions, data errors, and differences in clock rate make this algorithm less than totally accurate, but it has the pleasant property of working better as the load on the system increases because slot boundaries are synchronized more often. Retransmissions require host intervention, and hence an infinite series of retransmission collisions is avoided due to the essentially random variation in the hosts' response time.

MITRE

Several similar networks have been built by the MITRE Corp. The network described in [Hopkins 79, Holmgren 79] is microprocessor based and uses a CATV compatible medium. The low data rate of this system is due to the use of the microprocessor for all interface control functions.

The MITRE system's medium is composed of two coaxial cables joined by a "head end." The cables are unidirectional; one carries data toward the head end (inbound), the other carries data away from the head end (outbound). Signals on both cables are multiplexed using FDMA to create several channels; each channel can support a digital network, or can be used for video, speech, etc. The head end is basically a wideband amplifier that repeats all signals on the inbound cable onto the outbound cable. Each network interface is connected to both of the cables; the interface receives from the appropriate channel on the outbound cable and transmits on the appropriate channel of the inbound cable.

An interface with a message to transmit waits until the outbound channel is free and transmits on the inbound channel. After some propagation delay, the transmission arrives at the head end and is repeated onto the outbound cable system. Carrier sense, collision detection, collision consensus enforcement, and the other Ethernet access control mechanisms are used; they are slightly less effective due to the lengthier propagation delay.

Because of the unidirectional signal path, the MITRE system can replace the inbound and outbound cables with a cable "tree." Each "node" is an active repeater. On the inbound side, the repeater combines two or more input signals; on the output side the repeater splits and amplifies the signal. Using this type of structure, the size of the network is limited only by increasing propagation time and the consequent loss of bandwidth allocation efficiency; the network can easily be as large as existing CATV systems, which rarely are larger than 20 miles in radius.

GMAD network

General Motors Assembly Division (GMAD) has adopted cable bus technology as a cost effective, centralized method for providing diverse communications services within 12 of its 20 assembly plants [Smith 79]. The three generations of systems in use are built using CATV technology; separate FDMA channels support different applications including digital data collection and control links, audio communication, closed circuit TV, and camera control.

The main emphasis so far seems to be on replacing a variety of communication systems with a single cable bus in order to reduce communication hardware and maintenance costs. The digital channels are all low speed (less than 48 Kbps); cable channels have merely replaced dedicated point-to-point and multidrop lines.

Future plans for these systems include the development of integrated data networks which would couple all of the assembly plants to central computers in Detroit.

The major needs described in [Smith 79] are more reliable CATV cable hardware, maintenance, and repair procedures that can be performed by unskilled personnel, and the development of terminals and other cable stations that can be powered and controlled by the cable.

HYPERchannel

The HYPERchannel is a local network marketed by Network Systems Corp. of Minnesota [Thornton 75, Thornton 79]. It is designed to address a large computer center's need for high-speed data transfer between multiple CPUs, secondary storage systems, and other high-speed peripherals. The HYPERchannel medium is up to 1500 feet of coax, and runs at 50 Mbps. The network interfaces, called adapters, interface to the host at the host's channel level.

Adapters are controlled by a custom bipolar microprocessor, and include a channel interface customized to the attached host or peripheral, a data buffer of 1-8 K bytes, and medium interfaces for up to four coax links. Because of the high speeds involved, the microprocessor doesn't participate in data transfers; the microprocessor initializes and monitors transmissions and executes the adapter protocols.

The adapter implements two levels of protocol. The first (lowest) layer handles access control for the medium. The second level protocol implements buffer reservation and flow control.

The intent of the low-level protocol is to allocate bandwidth according to adapter priority; a high-priority adapter should be able to acquire as much bandwidth as it can use, regardless of the needs of lower priority adapters. When a transmission takes place, regardless of its priority, bandwidth for an acknowledgment is allocated. When the network is lightly loaded, allocation should be prompt; under heavy load, the network throughput should asymptotically approach the bandwidth of the medium.

The low-level protocol uses a mixture of priority reservation and contention to meet these objectives. Every data transmission synchronizes the adapters and is followed by three time periods: an acknowledgment period, a priority period, and a contention period.

The acknowledgment period immediately follows the data transmission. The acknowledgment period is intended to guarantee bandwidth for immediately available acknowledgments. During this period, the adapter that received the data transmission can send an acknowledgment transmission. This period ends after the acknowledgment transmission or after a timeout if no acknowledgment is generated.

The priority period follows the acknowledgment period. The priority period is divided into intervals; each interval is associated with a particular adapter. Each interval is approximately equal to twice the round trip time of the medium. During the priority period, an adapter can initiate a transmission only during its interval, and only if all previous priority intervals were unused. Any data transmission restarts the period mechanism. The ordering of the intervals thus imposes the desired priority of access to the medium. Under conditions of heavy network load, the priority system avoids collisions and improves throughput.

If no adapter initiates a data transmission during the priority period, the contention period begins. During the contention period, any adapter can initiate a transmission; hence all adapters contend for the medium. The intent is to minimize delay in accessing the medium during periods of light network load. Any transmission restarts the period mechanism, so series of collisions shouldn't be a problem.

The second level protocol handles buffer reservation in both transmitting and receiving adapters, and includes a packeting protocol to allow the transfer of arbitrary length data blocks through the adapters' limited buffers. Several varieties of adapter-to-adapter messages are defined to perform the necessary control functions.

Several difficulties were encountered with the original adapter protocols [Donneley 78a]. Some of these problems were simply design errors that were corrected by bug fixes, tuning, and new algorithms; two of the problems were associated with interactions between protocol layers. These two problems, retransmission failure and priority reversal, are described below.

The retransmission failure problem caused two-thirds of all data transmissions to suffer collisions when the system was heavily loaded. The cause of this problem was that the second level protocol couldn't request retransmission of unacknowledged data transmissions fast enough to use the priority period. Instead, all retransmissions coincided with the start of the contention period. In periods of heavy load, this almost guaranteed a series of lockstep retransmission collisions following an initial collision.

This leads to several observations:

1. The problem can be solved by speeding up the setup of retransmission requests by the second level of protocol, or by detuning the timeouts used in the bottom level protocol. The "optimal" timeouts for the first level protocol (i.e., as short as possible) make it impossible for the microprocessor to execute the second level protocol fast enough.
2. Any reduction in the retransmission setup time is counterproductive unless it allows transmission during the priority period. Otherwise, faster retransmission simply synchronizes transmissions at the start of the contention period.
3. The retransmission timing problem is worst for the highest priority adapters. Hence, it is quite possible that high-priority adapters will get less throughput because of more frequent collisions.

Two solutions were adopted for the retransmission problem. In new systems, retransmissions are handled by new first level protocol hardware that can meet the timing requirements. In existing adapters, the second level protocol was modified to create pseudo priority periods in the contention period. Software timeouts prevent retransmissions for a time proportional to the transmitter's priority. In essence, part of the priority protocol is transferred into the second level.

The second interaction problem, priority reversal, caused lower priority adapters to be able to achieve greater throughput than higher priority adapters. This would occur in situations where buffers were a scarce resource, and hence the allocation of matched transmit and receive buffers could be difficult. If a high-priority and a low-priority adapter had symmetric requests, i.e., each wanted to send a message to the other, they both allocate a transmit buffer, and then attempt to allocate a receive buffer in the other unit. Because the higher priority adapter attempts the receive buffer allocation first, due to its higher priority, it is first to notice the allocation conflict, and hence to abort and deallocate its transmit buffer. The lower priority adapter is hence able to complete its allocation.

The solution adopted was to associate priority with each resource, and to change the request mechanism to allocate resources in priority order with preemption of lower priority allocations.

The HYPERchannel experience suggests the following caveats to the designers of new high-level interfaces:

1. Some concepts, like that of priority in the HYPERchannel, transcend any single protocol layer, and hence must be known in several layers to be effective.
2. Tuning the timeouts and operation of any protocol layer to the constraints of that layer may adversely affect the performance of the total protocol system. This is especially true in systems with dedicated microprocessor or hardware control elements, because these tend to operate in a highly periodic manner without the random delays characteristic of large scale computer systems. Unfavorable timing combinations are thus either very rare or very common.
3. Efficiency and performance tuning requires the consideration of real-time event sequences; the abstractions used to model protocol layers often discard this information.

Consortium Ethernet

In December of 1979, Digital Equipment Corp., Intel, and Xerox announced a joint effort to develop a networking system based on the Ethernet [DEC 80]. The immediate aims of this effort are the production of a VLSI Ethernet controller, a corresponding standard, and a licensing policy that will promote wide use of the standard. According to the companies, this new Ethernet design is to be known as the production Ethernet, or simply the Ethernet; the original Ethernet is to be known as the "research" Ethernet.

Given the large amount of experience with the original Ethernet, it is interesting to note which parts of the original Ethernet design have been retained, and which parts have been changed. The major points discussed in the standard [DEC 80] are:

1. The new Ethernet will use the same contention ideas for access control.
2. The new system runs at 10 Mbps as opposed to the 3 Mbps rate of the existing Ethernet. The transceivers and analog signalling hardware are different.

3. The new system uses 48-bit source and destination addresses in the basic message format. The first bit designates whether the remaining 47 bits are a physical interface address or a multicast group identifier. One of the multicast addresses is reserved for use as a broadcast channel. This suggests some form of address recognition will be included on the Ethernet chip.
4. A 32 bit CRC is used instead of a 16-bit checksum, although a checksum is retained in new software protocols [Xerox 81a].

Salplex

Salplex is a multiplexed bus system developed by General Electric Ltd. which provides digital and analog data communication and power distribution over a coaxial cable [Smith 80a]. The system is designed to replace conventional wiring harnesses in busses, trucks, military vehicles, and eventually automobiles. A prototype system has been installed in a Ford Escort and is currently undergoing test. The eventual goal of this system is cost reduction by eliminating the conventional wire harnesses used in these vehicles.

The system uses the coax shield to distribute power with ground return through the vehicle's chassis. The equipotential field in the shield is claimed to reduce noise for data transmissions on the inner conductor.

Data transmission takes place between data units which also manage up to 8 power loads or sensors each. The typical automobile is seen as having approximately 40 loads and sensors, hence will require 5 or 6 data units. Data units are currently composed of 3 custom SSI CMOS circuits and 2 standard ICs, but will eventually be replaced by a single LSI CMOS IC.

Access control is a combination of contention and priority. When the bus is free, any data unit can seize control of the bus and transmit a packet that contains eight data words. Each data word either reports on an attached sensor or commands a remote load. Analog levels are communicated by varying the height of specific data bits. Once a data unit has transmitted, an integral timer prevents it from transmitting again for a period sufficient to allow all other data units to transmit. A priority mechanism allows designated loads, such as the break system, to achieve priority access to the bus.

2.3 CIRCULAR SYSTEMS

Introduction

This section discusses circular systems. The first subsection, "Common Design Principles," discusses techniques and problems which all circular systems have in common. The next subsection, "Background," contains a brief discussion of the historical evolution of circular systems. Following these sections, the LNI system is described in detail, and a brief overview of each of the other systems is given.

Common Design Principles

The common design elements uniting the circular systems listed in Table 2-2 are their use of a unidirectional medium and a circular topology. These systems must address many of the same problems as are addressed in bus systems (e.g., isolation, host interface), as well as some problems unique to their topology.

The main source of problems unique to circular systems is that a pure circular medium is unusable. At the signalling level, an unbroken medium wouldn't be unidirectional; the signal would propagate in both directions, resulting in multipath interference and endless propagation of every transmission. This implies the need to "break" the medium at each interface and to insert an interface-controlled repeater at the break. When the repeater is enabled, it enforces the unidirectional abstraction and makes the medium continuous. When disabled, the repeater provides artificial endpoints to the medium that can be used to sink existing transmissions and source new transmissions. The repeater function is typically implemented in the transceiver section of the interface.

The existence of repeaters creates several new concerns.

1. The transceiver receives Manchester encoded data and clock from the inbound side of the medium, and data, clock, and control information from the interface. It must use these signals to generate an outgoing signal. In order to do this, it must have some method for synchronizing the clocks of these signals.
2. Active repeaters add in-line delay to the medium; typically each interface will add 1 bit's worth of delay. This delay is significant with respect to the inherent propagation delay of the medium. For example, at 1 Mbps, a delay of 1 bit is equivalent to approximately 500 feet of medium.
3. The repeaters are active devices and as such are susceptible to failure due to power loss or hardware component failure. Because the failure of any repeater renders the medium discontinuous, and hence unusable, automatic bypass, backup, and other strategies may be necessary to insure adequate medium reliability.

Thus the circular medium is actually used as a set of linear media. In some systems the control of the medium continuity function influences the design of signalling and access control mechanisms; in others it is restricted to the signalling level.

The conceptual model of the medium is different in circular systems. Because of the inherent storage in the circular medium, several different bits of data are in circulation during transmission. The interface algorithms typically view the medium as two independent channels: a bit serial input stream and a bit serial output stream. At any given time, the interface must worry about processing the input bit (if any) and generating the output bit (if appropriate).

Another aspect of the conceptual model is the amount of buffering in the medium. In actual systems, the major element is the cumulative delay in all of the interfaces; the propagation delay is negligible. Some control algorithms require that additional synthetic delay be added to guarantee some minimum medium length.

Many different systems have been constructed to answer these concerns. In general, circular systems are much more varied in approaches to access control and medium management than bus systems. Four design choices are the main source of the differences; the choices are nearly independent and hence almost any combination can be found in in use. The choices are the following:

1. *What is the unit of medium allocation?*

   Two choices are variable size and fixed size message allocations.

   Slotted systems view the medium as a continuous "conveyor belt" of fixed size slots. Messages are constrained to be no larger than the slot size. This approach implies some bandwidth overhead. Medium bandwidth is wasted by the use of long slots to carry short messages; conversely a small slot size will increase the relative amount of message header per byte of data.

   Variable size messages are the more general approach and are usually more efficient in terms of bits of overhead per byte transferred. A variable length system requires either that the length of the data portion be included in the message header or that the length of the message is determined by some other means, such as carrier sense.

   Message chopping is a hybrid approach found in several systems. The basic idea is that the transmitting interface divides the host's message into fixed size transmissions; the receiving interface reassembles the message for delivery. Circuit-switching systems can be constructed using message chopping, a small transmission size, and some policy to avoid medium hogging by a single interface.

2. *How are messages written on the medium?*

   Transmit control in a circular system can be implemented in a manner analogous to the CSMA protocols used in bus systems. The only proviso is that the delay in the medium significantly increases the medium acquisition time and hence degrades performance somewhat. CSMA schemes are used in several circular systems, but only for the purpose of initializing the systems.

   The strategies in circular systems have been oriented in favor of schemes that allocate bandwidth in a controlled rather than probabilistic manner. Control information is passed in the medium's bit stream for this purpose.

   Slotted systems include slot formatting information and a bit which indicates whether the slot is full or not in each slot. To transmit a message, an interface waits for an empty slot. The message to be transmitted overwrites the current contents of the slot. Constraints are often necessary to prevent a hogging of the medium. These usually are implemented in the form of restrictions on the number of slots that can be used simultaneously by one interface or restrictions on the percentage of empty slots that can be filled by a given interface.

   Systems using variable length transmissions need a different form of control. A slotted system doesn't need to restrict the number of interfaces that can simultaneously transmit; this control is a byproduct of the limited number of available slots and the full bit indicators. In a variable length system, it is reasonable to assume that only a fraction of a message will fit on the medium unless the ring is artificially lengthened. Thus these systems will usually want to insure that only one interface transmits at a time.

   Token systems limit the right to transmit to the single interface that possesses the token. The token can be passed around the network by means of a control transmission. If the interface that has the token doesn't have a message to transmit, it passes the token to the next interface. If the interface possessing the token has a message to transmit, it transmits the message and then forwards the token. In an idle network the token circulates continuously; under heavy load, the interfaces get to use the medium sequentially. Token systems guarantee fair sharing of the right to transmit; in order to guarantee a fair share of bandwidth, restrictions on message length are necessary. In addition, precautions are necessary to insure that the token is not lost.

Another approach to controlling transmission rights is the insertion method. An interface using this idea formats the data to be transmitted into a shift register and waits for a boundary between messages on the medium. When the boundary is detected, the shift register is "spliced" into the medium. Depending on the removal protocol, the shift register is removed from the medium either when the original transmission returns to fill the shift register or when the shift register becomes empty.

The insertion method has been used in both fixed and variable size message systems. The design of such a system is very simple when a slotted protocol and return removal are used; otherwise a variable length FIFO buffer is required. The maximum length of the FIFO must be as large as the maximum size transmission.

3. *How are messages removed from the medium?*

In a circular system, messages must be removed from the medium by the action of some interface. There are two possibilities: the destination interface can remove the message as it arrives, or the interface that originated the message can remove it as it returns around the medium.

Removal at the destination has the advantage of removing the message as soon as possible. Because the message only uses some fraction of the medium's circumference, multiple simultaneous transmissions are possible when the transfer paths don't overlap. Although selecting compatible transmissions may be difficult, the aggregate throughput can exceed the medium's signalling rate.

The destination removal strategy also has its disadvantages. Each interface must buffer, and hence delay, the message until the interface can decide whether it recognizes the message address. This will add several bit times of delay per interface. If the address is corrupted by transmission error, the message may be prematurely deleted or, worse yet, may never be recognized and deleted. Similarly, the interfaces must be sure to never send messages to nonexistent addresses. Broadcast messages require that the message be forwarded by all but the last of the destinations.

Solutions to these problems usually involve a message field that is used to keep track of the age of a message; when the message gets too old it is deleted regardless of address. In order for this to work, there must be some way for this field to get regularly incremented, or the interfaces can use absolute timestamps and comparisons. One simple solution to all these problems is used by the TRW network: it periodically discards <u>all</u> data on the medium and restarts [Blauman 79].

When messages are deleted at the originating interface, the same identification problem exists. Because of this problem, source deletion is usually combined with a transmission control strategy that makes recognition simple. The simplest way is to guarantee that only one transmission can be in progress at a time. In this case, no ambiguity is possible; hence it isn't necessary to do address recognition, and hence there is no need to add buffering and delay.

Regardless of the exact algorithm used, source removal has one fundamental advantage: the source can verify correct transmission by examining the returning transmission as it is being deleted. If the transmission is damaged when it returns, then it was probably in error when it passed through one or more interfaces; retransmission is indicated. If the message returns undamaged, then the sender can assume that every interface on the network at least had the opportunity to process the correct message. The only way for this assumption to be invalid is for the message to have been damaged and repaired by compensating transmission errors. Further reliability can be added to this scheme by having all interfaces send negative acknowledgments in response to transmissions with bad CRCs.

4. *Is medium control centralized or distributed?*

This question distinguishes loops, which are circular systems with central control, from rings, which have distributed control. Depending on the system, control includes the functions of bit synchronization, message transmission, message deletion, and error recovery. Centralized control can be expected to predominate in situations where it is important to avoid duplicated function to reduce cost or in a system where only one host can or should exercise control (e.g., a host accessing a loop of peripherals). Distributed designs avoid a single point of failure and are advantageous in situations where the remaining parts of a system wish to continue to function in the presence of failed components. Because recovery may entail automatic reconfiguration, these systems should be insensitive to the number and configuration of interfaces. Any subset of the total system's components should be able to cooperate in system initialization, normal operation, and error detection and recovery.

Slotted systems are often centralized. The central controller, which may also provide a host interface, generates the bit timing and slot formatting for the loop; individual interfaces extract their clock signal from the medium and assume that slot formats will be present and correct. Only the central controller need worry about problems such as transmission errors, error recovery, and out-of-phase bit synchronization between its input and output signal.

Centralized control can extend to the allocation of transmission rights. In *polled* systems, a central controller explicitly allocates the right to transmit to the other interfaces on the loop. The central controller can also delete messages.

Token systems rely on the existence of a unique token to guarantee that transmissions don't collide. The token is central control in a sense, but tokens can be easily replaced; hence these systems are suitable for distributed control. Distributed control in a token system is only a problem during token initialization and error situations; in these cases control can't be allocated by the token. For example, if all interfaces detect token loss simultaneously, they may all try to create a new token simultaneously and fail due to interference. Randomized retries solve this problem.

Insertion systems have been built with both centralized and decentralized control. Variable length insertion systems usually require complicated enough transmission systems so that there would appear to be little gained by centralization.

Background

Pioneering work in circular systems was done by three groups at Bell Labs.

The first of these is a token scheme described by Farmer and Newhall in [Farmer 69]. As a result, token systems are often called Newhall loops. The system used distributed control for normal operations but included a "loop supervisor" node for timing control and error recovery. Out-of-band signalling (i.e., special analog levels with no meaning in the digital bit stream) is used to mark message boundaries and to pass the token.

The second effort was reported in [Pierce 72a] and [Pierce 72b]. This was a slotted system with destination removal. As a result, slotted systems are often called Pierce loops.

The third system is Spider [Fraser 75]. This system has a very centralized design: the loop is managed by a controller that includes a Tempo Corp. minicomputer. Messages between hosts on the loop are transmitted to the Tempo and then forwarded to their ultimate destination. This system was used in production as early as 1972, and grew to "11 computers of 5 types" by 1975.

These systems led to several other circular systems. The following sections discuss in detail the family of interfaces that started with the RI at the University of California at Irvine [Farber 72b]; following this discussion, the other circular systems from Table 2-2 are summarized.

DCS RI, UCI and MIT LNI, and NS LNI

This section describes a family of interfaces. The three different interfaces are used in different networks: the Ring Interface (RI) is used in the Distributed Computer System (DCS); the UCI and MIT Local Network Interface is used in the MIT Laboratory for Computer Science and elsewhere; the NS LNI is used in a network at UC Berkeley. The discussion first traces the evolution of features, and then covers the operation of the most recent member of the family in detail.

DCS RI

The Ring Interface (RI) was designed to support message communication for the Distributed Computer System (DCS). The DCS uses messages for all forms of interprocess communication and uses process names rather than location to specify destinations. Accordingly, the RI has full-duplex DMA data paths and a prompt acknowledgment facility to support the expected heavy use of messages and provides a mechanism for recognizing process names in the interface hardware.

The RI transfers messages around a ring composed of two twisted pairs at 2.2 Mbps. The message format used by the RI is shown in Figure 2-4. The Origin Process Name (OPN) and Destination Process Name (DPN) are each 16 bits in length, corresponding to the length of process names in DCS. Destination addresses are recognized by the name table, an associative store in the RI. The name table has space for 16 names. The acknowledgment feature is implemented using fields appended to the basic message during transmission. The detailed operation of these features is described in the LNI section.

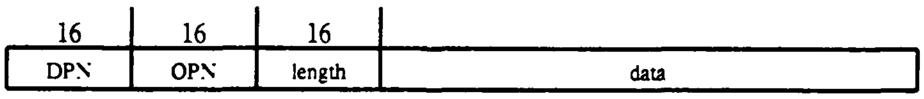


Figure 2-4: RI message format

UCI and MIT LNI

In 1976, an effort began at UCI to develop a new interface based on the RI. The new interface, called the Local Network Interface (LNI), was designed to operate with the PDP-11 and to satisfy the following goals:

1. It was expected that the LNI would eventually be implemented as an LSI chip or chip set. Thus although the actual implementation was done in TTL, the design was based on LSI constraints and capabilities, rather than those of TTL. The major impacts of this decision were the use of bit serial data paths throughout the interface and a modular structure composed of many small state machines controlled by PLAs.
2. The 16-bit size of names was felt to be too small. Names were lengthened to 32 bits.
3. Experience with DCS and planning for new types of hosts suggested that it was difficult to predict the number of name table entries that a system would need. Whereas a large host or gateway might need many names, a less capable host such as a terminal would only need a name or two. Given the LSI goal, the name table was designed so that a minimal name table could be implemented on the LNI main chip and extended indefinitely by the addition of a chain of name table expansion chips external to the main chip. Eight names were built in each of the TTL interfaces.
4. Construction of broadcast channels required the use of one name table entry per channel per process using the RI scheme. The ability to mask off portions of the name comparison was added. Masks were included in both the message text and in the name table entries.
5. The host interface section of the LNI was designed to provide a clear partition between the part specific to the PDP-11 and the part deemed necessary for any host. An attempt was made to generate a small yet general interface between the two.
6. Because the LNI would eventually be used with a variety of hosts, it was not clear how much buffering would be required in the DMA data paths, or whether DMA interfaces would be possible in all situations. Hence, the LNI was designed to include an easily expandable FIFO in both the receive and transmit data paths.
7. The line signalling protocol used by the RI included separate clock and data lines and used out-of-band signalling. The LNI was designed to use a single Manchester encoded signal.
8. The analog signalling system was redesigned to provide ground isolation between LNIs and to improve the permissible distance between interfaces. A data rate of 1 Mbps was adopted for the prototype.
9. Comparison between the functionality of the Ethernet interface and the RI revealed that a great deal of commonality exists. An attempt was made to isolate and modularize the differences so that the LNI might possibly be used to implement a contention protocol.
10. Use of the DCS system had revealed that the prompt acknowledgment protocol was inadequately protected against transmission errors. Error detection facilities were added.

This version of the LNI was designed and initially debugged by UCI and was adopted by MIT's Laboratory for Computer Science (LCS) for its local network. It is currently in use at LCS and at UCLA. The LSI version was never built.

NS LNI

The third generation of this family is the product of Network Systems (a California company, not the Network Systems Corp. which built the HYPERchannel). The design differed from that used in the UCI and MIT LNI in the following ways:

1. The transmission speed was increased to 2 Mbps.
2. The PDP-11 interface, particularly the DMA, was redesigned to simplify host programming. allow for the higher data rate, and eliminate several persistent problems in UNIBUS arbitration.
3. In the RI and first LNI, the name table binds message address specifications to hosts. This operation is then essentially repeated by the host in order to bind the message to a process's connection state information.

   In order to unify these operations, each name in the associative memory was expanded to allow storage of a pointer to the connection state information. Facilities were added to allow the host to retrieve this information following message reception, and to allow the host to query and maintain this information.
4. The default name table size was increased to 16 names.

A block diagram of the LNI is shown in Figure 2-5. The dotted lines partition the LNI into its main sections. The top section is the medium interface. The input and output sides of the LNI are the middle sections on the left and right, respectively. At the bottom of the figure, the host interface is partitioned into the DMA and the status and control sections. Only the main data paths are shown; with the exception of the paths between the FIFOs and the DMA and the connections to the UNIBUS, all paths are bit serial. The following sections describe these sections and the use of the LNI in greater detail.

LNI Medium

The LNI medium and medium interface are shown as the top section of Figure 2-5. The medium is twisted pair.

The line receiver is a high-speed opto-isolator. Opto-isolators are current mode devices that are powered by the incoming signal, so the matching line driver is selected for high current drive. Isolation of 3000 volts is guaranteed by the opto-isolator; each interface uses the local system ground as its reference for transmissions. A Manchester code is used for signalling.

The central element of the medium interface is the clock extractor and bit selection logic. This logic looks to the medium likes two bits of delay. The first bit of buffering is available to the input side of the LNI for inspection or modification; the second bit is for use by the output side. In order to make these bits useful, the medium interface generates clocks for the input and output sides which are synchronized with the respective data bits when these bits are present and run free otherwise.

These clocks obey the following conventions:

1. When no data is present on the ring, the input and output clocks should still run so that the LNI can respond to host commands.
2. When a transmission arrives, the input side clock should synchronize to the incoming data.
3. When the transmission is to be repeated through the LNI, the output clock must synchronize to the input clock.
4. When the LNI originates a transmission, the output clock must run at 2 Mbps with as

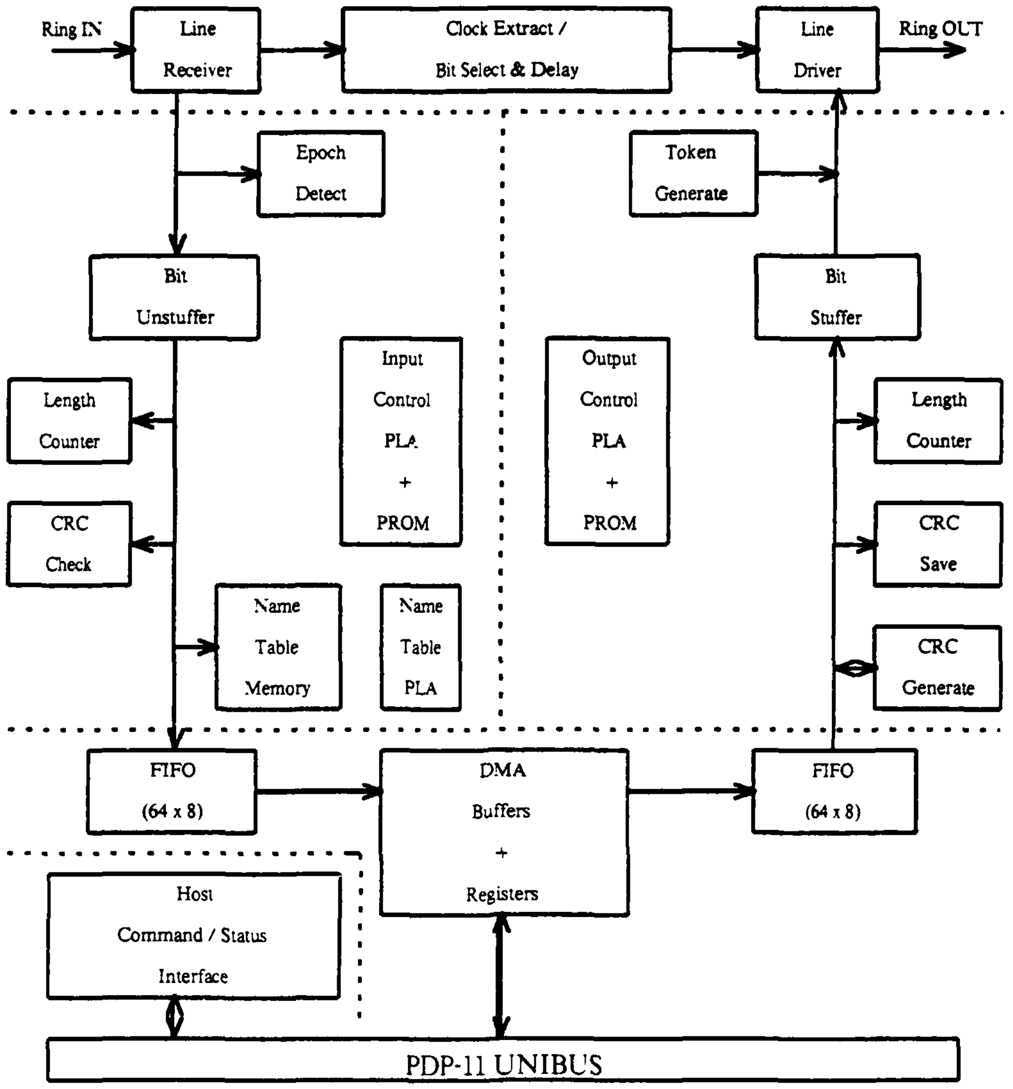


Figure 2-5: LNI block diagram

constant as possible a rate, while the input clock must track the returning data stream. The returning data will probably be out of phase with respect to the transmit side, and will also exhibit phase jitter.

Although it would seem to be simpler to use a single clock synchronized to a continuous data stream, that approach has practical difficulties. If the data stream frequency is controlled by a single reference, then all other units can phase lock to the reference, and everything will work. However, this creates a unique node and a single point of failure. If no reference is used, then the frequency of the signal will tend to drift due to noise and component differences. Any attempt to limit the amount of drift will cause bit loss when the limits are reached. The adopted scheme resolves these difficulties by letting each interface refer to the frequencies of its own transmissions; the practical difficulty with this approach is insuring a smooth transition between references.

The data selection and clock control system of the LNI consists of a single PLA controlled state machine driven by a 16 MHz crystal controlled clock.

On the input side, the 16 MHz clock allows the PLA to sample the input data stream at 8 times the data rate, or 4 times the signalling rate of the Manchester code. The state machine simulates a phase-locked loop to insure accurate data recovery in the presence of frequency and phase jitter. When no input carrier is present, the input side is supplied with a clock generated by dividing down the 16 MHz reference.

The output clock is generated by dividing down the 16 MHz reference. When the output side is transmitting, or no input carrier is present, the 16 MHz clock is divided by 8 to provide a stable 2 MHz clock. When the output side of the LNI is functioning as a repeater, the output clock must track the frequency of the arriving signal. When sampling detects that the input side is starting to get ahead of the output, then the output divisor is set to 7 to speed up output. A slow input side is compensated for by using an occasional divisor of 9.

This scheme avoids the drift problem because clock is independently generated for every transmission; even if drift occurs, it does so for only one trip around the medium. The only exception to this is a token that circulates continuously. This turns out not tc be a problem because of the short length of the token.

LNI Access Control

The LNI access control protocol is designed to send messages around the ring using the medium as a simple bit serial channel. The LNI that originates a message also deletes it. A token passing system insures that only one LNI transmits at a time, although the number of messages in transit is limited only by the length of the ring. If the ring is very long, messages circulate as a connected train that is terminated with the token. If the ring is short, the train can hold only a fraction of a message, and the head of a message will be deleted before the tail is finished being transmitted.

Two 10-bit synchronization patterns signal message boundaries. Both patterns start with a unique bit pattern (8 ones) that is called the epoch. In order to insure that epochs don't occur in data, all messages are encoded using a bit-stuffing protocol similar to IBM's SDLC [Donan 74]. The bit after the epoch distinguishes between the two synchronization patterns. The last bit of the synchronization pattern is always zero so as to reset the bit-stuffing system.

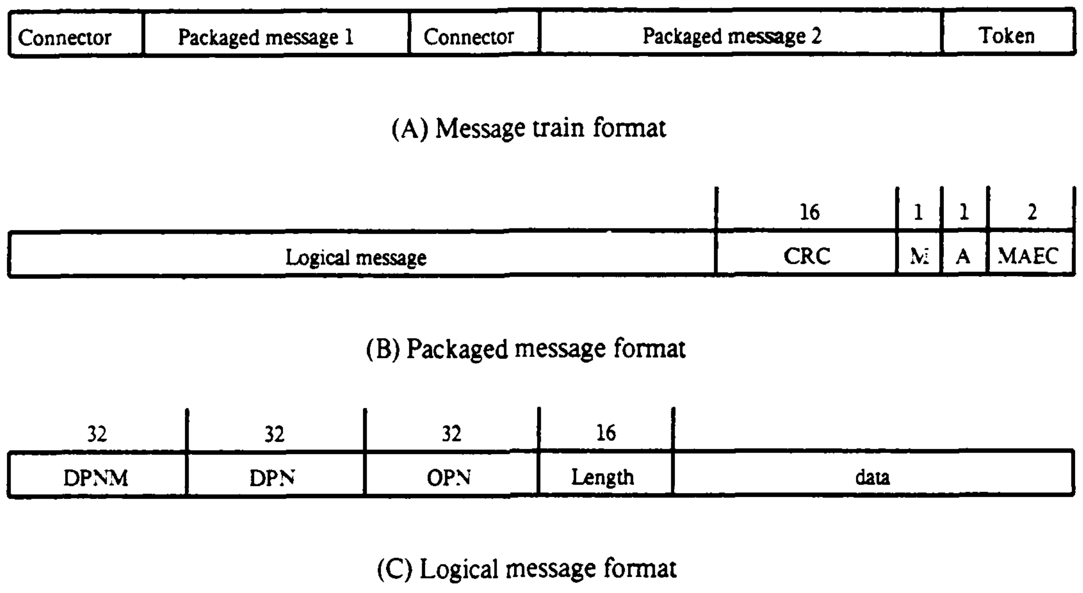
The first of the synchronization patterns is called the connector. A connector is present before all messages and signals the start of a message.

The other synchronization pattern is the token. The token denotes the end of a message train. Figure 2-6 (A) shows an example of a message train consisting of two messages; if no messages are present, the train is simply a token.

When an LNI wishes to transmit, it waits for the token to arrive. As the token passes the LNI that wishes to transmit, the output side changes the bit that distinguishes the token from a connector. Following the newly created connector, the LNI outputs the message and packaging information described below in the "LNI Buffering and Formatting" section. After the packaged message, the output side generates a new token.

No LNI can be indefinitely locked out of the medium by heavy traffic load. The maximum time that an LNI has to wait for the token, and hence the right to transmit, is equal to the time required to complete one ring circuit plus the time required for all other LNIs to transmit one maximum length message. Although the maximum size message the LNI will transmit is 64K bytes, a much smaller maximum will usually be required by the host software to simplify buffer allocation. The DCS system has 3 hosts, a 1 kilobit maximum packet size, and negligible circuit time. At a 1 Mbps data rate, this yields a worst case access time of 2 milliseconds.

When the output side begins to transmit, it disables the repeater between the input and output sides of the LNI. The repeater stays off until the input has discarded one message, then it is reenabled.



(A) Message train format

(B) Packaged message format

(C) Logical message format

Figure 2-6: LNI message formats

<u>Transmission error handling</u>

Errors in the transmission system may destroy or create connectors and tokens, or alter message data.

The simplest error is one in which the token is destroyed. This condition is identical to that encountered when the ring is first started.

When a host queues a message for output, it also starts a watchdog timer. If the token is destroyed, the message is never output, and the timer goes off. The host then issues a special output command to the LNI that forces transmission to begin without waiting for a token.

Note that several LNIs may attempt to force the token simultaneously. If this happens, the tokens will either destroy each other or start two separate message trains.

If the tokens destroy each other, the timeout sequence repeats. If the timeouts are different in each host, the token will eventually be restarted. Essentially, the LNI operates in contention mode until the token is restarted.

The situation in which multiple trains coexist is resolved by the same mechanism that deals with damaged connectors. When a message is transmitted, the LNI that transmitted the message attempts to verify that the message it deletes is the same as the message it transmitted. If not, it deletes two messages.

Without comparing the entire message text, it is impossible for this comparison to be totally accurate, and it is undesirable that token restart be invoked every time a transmission error occurs. Instead, the comparison is restricted to comparing the CRC transmitted with the message against the CRC in the returning message. (Actually, the check also depends on the accuracy of the message length field in the returning message since it is used to identify the position of the CRC in the returning message.)

If the CRCs don't match, or if the returning message contains an unexpected epoch, the double deletion mechanism is invoked.

LNI Buffering and Formatting

As illustrated in Figure 2-6, the LNI can be thought of as having three levels of message format:

1. The message train format
2. The packaging format
3. The logical message format

The message train format was described in the LNI access control section. The other two formats and the buffering system are described in this section.

The logical message format

The logical message format is composed of the message fields that are seen by the sending and receiving hosts; it does not include the packaging or access control fields added to the message by the LNI during transmission. These fields are shown in Figure 2-6 (C).

The data field is the message text. The LNI places no restriction on the number of bytes in the data field other than the maximum byte count that can be represented in the message length field (65535 bytes).

The Origin Process Name (OPN) is the name of the process that sent the message. The Destination Process Name Mask (DPNM) and Destination Process Name (DPN) fields specify the destination of the message. The DPN and DPNM fields are compared against the name table of every LNI as the message circulates around the ring. Those LNIs that recognize the destination attempt to copy the message as it passes.

When a DPN and DPNM are compared against a name table, a **match** condition is recognized if one or more of the name table entries match the DPN and DPNM combination. Each name table entry is also composed of a 32-bit name and a 32-bit mask. Comparison is done one bit at a time, using the corresponding bits from each of the four fields. For **match** to occur, all bit positions must match. For a given bit position to match, either the mask bit must be on in either the DPNM or the name table mask or the name bits must be equal.

In the case where a message is addressed to a unique process name, both of the mask fields will be zero. A message whose DPNM is all ones will match any name table entry. A name table entry with a mask of all ones instructs the LNI to attempt to copy all messages on the ring.

The packaging format

The packaged message format is shown in Figure 2-6 (B). A packaged message consists of the logical message supplied by the host and fields added by the LNI to detect transmission errors and to implement the prompt acknowledgment facility.

CRC

The CRC is a conventional 16-bit CRC that will detect all single-bit errors and most multiple-bit errors. The CRC field is generated by a 9401 chip in the output side of the LNI ("CRC generate" in Figure 2-5) whenever a message is transmitted. The CRC covers the logical message fields. The input side uses another 9401 to check the CRC of all arriving messages ("CRC check" in Figure 2-5).

The input side compares the CRC it calculated with the CRC imbedded in the message. The result of this comparison is reflected in the input status presented to the host with the message. If the comparison failed, the usual host action is to discard the message.

When the output side transmits a message, it uses the input side CRC comparison for the deleted message to determine whether the transmission is successful. The result is presented in the output side status that accompanies completion of the transmission. If an output side CRC error is detected, the usual host response is to retransmit the message.

Low-level acknowledgments

The **match** and **accept** bits in the packaged message format implement the prompt acknowledgment facility of the LNI. They are shown as M and A in Figure 2-6 (B). Both of these bits are set to zero when the message is transmitted.

As the message circulates around the ring, it is processed by all of the input sides of all of the LNIs on the ring. Each of these input sides matches the DPNM and DPN against the entries in the attached name table. This matching operation occurs for all messages on the ring, regardless of whether the input side has been enabled or not. If the input side is enabled, and the DPNM and DPN match at least one of the name table entries, then the LNI attempts to copy the message. The copy attempt may fail for several reasons: bad CRC, DMA overrun, unexpected token, etc.

Input sides that recognize the message address use the **match** and **accept** bits to signal the success or failure of the copy attempts to the transmitting LNI. Input sides set the **match** bit if they recognize the DPNM and DPN, but can't copy the message. Input sides set the **accept** bit, if they recognize the DPNM and DPN and can successfully copy the message. Note that more than one LNI may want to set either of these bits; the setting operation is a logical OR.

When the message completes its trip around the ring, the state of the returning **match** and **accept** bits is recorded and reflected to the host with the output status.

The host interprets these bits to determine the results of the transmission. The four possible cases are shown in Table 2-3.

Table 2-3: Match/accept results

Match	Accept	Meaning
0	0	The message was addressed to a process that does not exist; no LNI recognized this message.
0	1	The message was successfully transmitted to one or more processes; at least one LNI recognized and copied the message.
1	0	No process received the message; however, at least one LNI recognized the address in the message.
1	1	This message was addressed to processes in at least two hosts; at least one LNI was able to copy the message, and at least one LNI was unable to copy the message.

Typically the host retransmits messages that return with the **match** set. Duplicates are eliminated in the host by the sequencing protocol.

Match/accept error processing

The **match** and **accept** bits provide a reliable method of driving the transmit sequencing protocols when transmission errors are absent. The possibility of transmission errors demands that further steps be taken to insure proper interpretation of the **match** and **accept** bits.

When a returning message is damaged, the **match** and **accept** bits must be ignored. For example, a transmission error may have altered the message address. The damaged address could have resulted in the message missing a desired destination and being copied by an undesired destination. No harm is done because the improper destination will discard the message due to the CRC error, and the source will retransmit.

A more difficult problem is raised by the possibility that the **match** and **accept** bits may themselves be damaged by transmission error. In the original RI, this was a problem. Accordingly, the LNI design goals included the protection of these bits.

One alternative was to move the **match** and **accept** bits in front of the CRC, so they could be protected by the CRC. This would mean that the CRC would have to be recomputed every time the **match** and accept bits were altered.

One problem with this approach is the possibility of a damaged message. In this case, the LNI shouldn't change the CRC to a correct value; this would destroy the effectiveness of the CRC mechanism. The only appropriate choice is to force the CRC to stay incorrect.

Unfortunately, the CRC error isn't detected until the last bit of the CRC is processed. Hence, either we reduce the effectiveness of the CRC to one bit in certain cases, or we must increase the amount of buffering, and hence delay, in each LNI to allow modification of more CRC bits. These choices were unacceptable.

The adopted strategy was to introduce an error check field specifically for the **match** and **accept** bits. (MAEC in Figure 2-6 (B)) The MAEC field is the **match** and **accept** bits in complemented form.

When an LNI changes the **match** or **accept** bit, it also changes the appropriate bit in the MAEC field. If the MAEC field is inconsistent with the **match** and **accept** bits, the update algorithm creates a new inconsistent state.

This system catches all single-bit errors, and all errors due to a burst of ones and zeroes. Other burst errors will probably destroy the token or connector that must immediately follow the MAEC field and result in a detectable framing error. While this system is less than perfect, it represents an improvement of several orders of magnitude over the unprotected **match** and **accept** system.

LNI buffering

Aside from the single byte of buffering necessary for serial-to-parallel conversion, message data is internally buffered by the LNI in two symmetric FIFOs: one for input, and one for output.

These FIFOs partition the data handling functions of the LNI. On the host side of the FIFOs, data transfer is parallel and oriented towards the asynchronous behavior of the UNIBUS; on the medium side, data transfer is synchronized to the flow of data on the medium.

The FIFOs also separate two control domains. The medium side of the interface "the LNI side," and the host side of the interface "the DMA side," each have independent control and status interfaces to the host and don't coordinate their internal activities. The separation is complete in regard to the abstract operation of the LNI, and nearly so with respect to the actual implementation.

Message reception

Message reception is controlled by the input LNI and the input DMA. When enabled by the host, the input side waits for a message whose DPNM and DPN are recognized by the name table. Unfortunately, the input side has to receive the first eight bytes of the message (the DPNM and DPN) before it can know whether the message is to be received or ignored. When enabled, the input side buffers the DPNM and DPN of all messages in the FIFO; if the name table fails to recognize the DPNM and DPN, this information is purged. Until this decision is made, the DMA is inhibited from accessing the FIFO. If the DPNM and DPN aren't recognized, the FIFO is cleared and the input side waits for a new message. If the DPNM and DPN are recognized, all following data bytes are copied into the FIFO until either the message length is exhausted or an error is detected.

As the input side copies the message, it enters data bytes into the FIFO at 4 microsecond intervals. If the message size exceeds 64 bytes, the FIFO can't hold the complete message, and hence the host must empty the other end of the FIFO before an overrun occurs. The rate at which data must be removed from the FIFO is constrained by message length and the latency of the removal process in response to the arrival of a message. Basically, the FIFO allows the removal process to trail the arrival of message data so long as the lead never exceeds 64 bytes.

On the host side, data can be removed using either of two methods: programmed I/O and DMA.

Programmed data transfer requires host intervention for each byte of data that is transferred; the host acquires the next data byte by referencing the input data memory location. This approach is usually unsatisfactory due to its low data rate and the large amount of host attention required.

DMA transfer requires host intervention only for the initial setup; the host specifies a starting address and a maximum length before enabling the DMA. The DMA then attempts to empty the FIFO into the specified buffer. If the DMA gets ahead of the message data, it simply waits until a new byte becomes available through the FIFO.

Note that in each transfer method the completion of message reception is signalled by the LNI side of the interface when the last byte of the message has been placed in the FIFO; the message may not be completely transferred into the host's memory until some time later.

Message transmission

On the transmission side, the roles of the host and the medium are reversed; the host typically fills the output FIFO before initiating transmission, and then attempts to keep the FIFO full until the complete message is transmitted.

The ability to preload the FIFO before initiating transmission means that the host software can obtain the message header from a separate source rather than requiring it to precede the data section of the message. This ability can often be used to avoid needless copying of data into system buffers.

LNI Host Interface

The LNI hardware that implements the host interface is divided into two sections: the memory locations supported by programmed I/O, and the hardware used to generate interrupts and perform DMA. The UNIBUS protocols follow a similar split in that all UNIBUS activity involves a master device and a slave device. The master is responsible for gaining access to the UNIBUS and controlling the transfer. The slave merely obeys the commands issued by the master. When the LNI performs programmed I/O it is the slave; DMA and interrupts require the LNI to become the bus master. Hence the programmed I/O part of the LNI is minor in terms of complexity and part count; interrupts and DMA are the major part of the interface cost.

From the viewpoint of the host program, the programmed I/O locations are the only visible parts of the interface; DMA and interrupts occur asynchronously and are seen only by their side effects. The programmed I/O registers are divided into five sets in a manner that parallels the logical organization of the interface:

1. Input LNI
2. Output LNI
3. Input DMA
4. Output DMA
5. Name table

The input LNI section contains two registers: a command and status register (CSR), and a data register. The LNI input CSR contains command bits that allow the host to enable message input and input interrupts, and status bits that report errors and the current state of the input side. The LNI input data register allows the host program to access the input FIFO.

The input DMA contains a CSR to control DMA activity, as well as registers to set the input DMA address, length, and the interrupt vector for the input side.

The LNI output and output DMA register sets contain the analogous CSRs, data register, and address specifications.

The name table interface allows the host to specify the names stored in the name table, and to associate connection state information with each name. Each entry contains the following components:

1. A 32-bit mask
2. A 32-bit name
3. A 64-bit user field that contains the information to be associated with this name
4. A full bit that denotes whether this entry is in use
5. A select bit
6. A mark bit that is set in all entries which matched the last message received by the LNI

The host software accesses name table entries in programmed I/O mode through a 16-byte name table window and the 1-byte name table CSR. By means of commands issued to the name table CSR, the host causes name table entries to be copied into the name table window and vice versa. While in the window, the data can be manipulated using standard processor instructions.

The host selects the entry that is copied into the window by means of one of the following commands:

1. Select by empty
2. Select by mark
3. Select by internal match (selects all entries that match the current contents of the window)
4. Select next

The first three of these commands set the select bits of all name table entries based on the named consideration. Following the selection process, a selected entry is copied into the window. The select next operation resets the select bit of the entry currently associated with the window and searches for another entry with a set select bit. Thus the select next command allows the host to scan through all selected entries after one of the first three select commands has selected a subset of the name table. Because any of these commands may fail to select an entry, a bit is provided in the name table CSR to indicate whether any entry is currently selected.

One use of this mechanism is the processing of an input message. In this case, the host software issues a select by mark to select all entries which matched the message. Using the attached connection state, the software can determine where to route the message. If the message is a broadcast, select next commands can be used to find all names that match.

The actual update of name table entries is handled by three other name table commands:

1. Set full
2. Set empty
3. Write back

The first two commands are used to control the setting of the full flag for the currently selected entry. The write back command writes the contents of the name table window into the currently selected name table entry. All of these commands are interleaved with the matching of names that takes place whenever a message passes the LNI. In order to insure predictable operation, name table commands are not processed while a message name is being matched against the name table.

Other Circular Systems

DLCN

The Distributed Loop Computer Network (DLCN) is a research effort at Ohio State University that includes the design of network hardware, the Distributed Loop Operating System (DLOS), the Distributed Loop Data Base System (DLDBS), and several other projects [Liu 78, Babič 77]. The planned prototype network is to include a 370/168, a DecSystem 10, and several minicomputers.

The proposed network interface is based on a 2900 bit-slice microprocessor and a variable length insertion protocol. The microprocessor controls transmissions and is designed to allow higher levels of protocol to be imbedded in the controller. The interfaces include a special variable length shift register that can be inserted into the medium.

The insertion protocol allows an interface to output a message between already circulating messages so long as the interface has sufficient unused buffering in its shift register to "expand" the ring to include the new message. Once transmitted, messages are normally removed by the destination interface. The exception to this is for broadcast messages, which are copied by all interfaces and which are removed by the transmitting interface after one circuit of the ring.

Message transmission uses up shift register space if another message arrives while the interface is transmitting. This space is reclaimed when the interface receives a message, or during periods in which there is no traffic from upstream. One of the advantages of this scheme is that the amount of space available in each interface's shift register tends to enforce a low-level flow control. Assuming that traffic among interfaces is fairly evenly distributed, an interface that is constantly transmitting will use up its shift register space and be forced to a lower level of activity, whereas an interface that is only lightly loading the ring will not be similarly restricted. Even the interface that wants to transmit heavily will not be restricted until free space on the medium becomes scarce.

The claimed advantages of the variable length insertion scheme are that a message can begin sooner, and hence complete sooner, and that this system delivers greater throughput than other ring systems. Simulation studies [Liu 78] support these claims based on a Gaussian message size distribution, although they do not distinguish between gains made possible by the insertion scheme and gains made possible by destination removal.

In any case, these simulations verify the intuition that destination removal will approximately double the throughput of a saturated system and that quite possibly the shift register insertion scheme allows for saturation to be maintained.

Under light load the situation isn't as clear. The simulation results assert that the DLCN system is superior, in terms of delay, at any load. But in very lightly loaded situations, it would seem that the additional delay introduced by the address recognition needed for destination removal would be greater than the gains by simultaneous transmissions. Using the DLCN message format described in [Liu 78], each interface will need to buffer 24 bits of data before it has the complete destination address available. The simulation studies are based on a 10 interface network, so an average delay of 120 bits seems inevitable. The delay in a token loop might be 3 bits per interface, hence a total latency of 30 bits and an average token latency of 30/2+token length or 25 bit times is to be expected. Hence whenever only one message is in transit the DLCN approach must lose.

Other disadvantages of the DLCN scheme are related to its operation in the presence of errors.

Destination removal requires some mechanism fcr the removal of messages whose address becomes unrecognizable due to transmission errors. The DLCN solution is to divide the ring into two halves. Each interface belongs to one of the two halves (say west vs. east), and all interfaces of one half are contiguous. A two-bit counter in the message format is initialized to zero when the message is transmitted, and a one-bit field is set to signify the origin half of the message. Whenever the origin bit is found to be different from the half identity of the interface that is currently forwarding the message, the field is toggled and the count is incremented. This scheme increments the counter whenever the message crosses an east-west or west-east boundary. When the counter reaches 3, the message is automatically deleted.

The second difficulty with the scheme involves the fact that the right to transmit depends on an interface's ability to empty its shift register and the fact that this ability is only indirectly related to the amount that the particular interface is transmitting. Asymmetric traffic patterns may result in an interface losing its ability to transmit, even if it is not sourcing much traffic. For example, consider a segment of a DLCN loop containing three consecutive interfaces. If the most upstream interface is sourcing heavy traffic directed at the most downstream interface, neither of these two interfaces nor any of the interfaces outside of the subset has any difficulty in emptying their shift registers. Yet the interface in the middle of the subset may have a great deal of difficulty. The DLCN solution to this problem is to allow each interface to transmit special flow-control messages to other interfaces that seem to be hogging bandwidth. These messages cause the addressed interface to cease transmitting for some period of time and hence allow other interface's shift registers to drain.

The available evidence suggests that this is at best an academic problem since few hosts could conceivably source enough traffic to make it an actual problem. However, the scheme to address it seems adequate, if a bit complicated.

Cambridge loop

The Cambridge loop is a local network built at the Computer Laboratory at the University of Cambridge in England. The network design began in 1974, and has progressed to a fully operational system that includes about a dozen hosts of eight types [Wilkes 79].

The network is based on a slotted loop protocol that includes two special interfaces: a monitor station that generates the slots and recovers from errors and a logging station that logs error reports from all interfaces on the loop. Messages between hosts are conveyed using a sequence of slots; once a message starts, the receiving interface ignores slots from all other interfaces. Each slot carries a fixed length packet consisting of the following fields:

1. A full bit that signals whether the slot is occupied
2. A monitor bit used to detect infinitely circulating slots
3. An 8-bit source address
4. An 8-bit destination address
5. A 16-bit data field
6. A 2-bit response field

Message transfer thus involves a high overhead. Wilkes [Wilkes 79] estimated that a 10 Mbps medium rate yields at best a 1 Mbps data rate.

When an interface wants to transmit, it waits for an empty slot. Upon finding an empty slot, the interface sets the full bit, fills in the source, destination, and data fields, and sets the response field to all ones. The slot circulates around the loop. Every interface sees the slot, and the interface that recognizes the destination can take three actions:

1. It can accept the data.
2. It can refuse the data because it is already receiving a message from another interface.
3. It can refuse the data because it is "busy," a condition that includes exhaustion of buffers, host not ready, etc.

The destination interface clears one or both of the response bits to distinguish among the three cases. Eventually the slot returns to the transmitting interface. The transmitting interface resets the full bit and passes on the returning value of the response code to the message layer. Interfaces depend on there being a fixed number of slots in the loop to detect the returning slot; hence most slots get deleted even if transmission errors occur. In order to fairly distribute the loop's bandwidth, interfaces are prohibited from refilling returning slots.

In some of the Cambridge interfaces the message-chopping protocol is included in the interface. In those interfaces an 8X300 microprocessor interfaces to the host and performs these tasks. In simpler interfaces, the host is interrupted after every transmitted and received slot. All interfaces include an address register that determines which slots are accepted and which are refused. If the address register is all ones, then slots from any source are accepted; otherwise only slots from the specified address are accepted, and all others are returned with a refusai response code.

Several mechanisms are included in the Cambridge system to enhance its reliability and maintainability:

1. The loop medium consists of several wires. In addition to the data bits, power and bypass signals are distributed. The separate power lines allow the loop to function even if the host loses power. The bypass signals allow the monitor station and other control points to reconfigure failed interfaces out of the loop.
2. As an added check to the full slot deletion mechanism, the monitor slot sets the monitor

bit in every slot that passes through the monitor station. If the monitor station receives a slot whose monitor bit is set, it resets the full bit of the slot and reports the error to the logging station.

3. Whenever any interface detects a slot with bad parity, it corrects the parity and sends a single slot error message to the logging station. The logging station can deduce the point in the loop at which the error originated by examining the source address of the error report.
4. Whenever an interface fails to receive slot formatting information, it transmits a continuous stream of slots to the monitor and logging stations. Thus if the loop becomes discontinuous or if an interface fails, the monitor station, or any interface downstream from the break, can determine the location of the failure by noting the identity of the interface that is sourcing this stream.

The interfaces without the onboard message buffer consist of a 16 IC transceiver and a 66 IC main interface. The Cambridge loop interface is currently being reimplemented using the Ferranti uncommitted logic master slice chips [Smith 80b]. When complete, these LSI parts will allow a Cambridge interface to be constructed using 2 LSI chips and 2 external shift registers. A commercial version of the interface will reportedly cost $2000 per interface for a 15 station system [Smith 80b].

PRIMENET

PRIMENET is a token-based ring network built for use with Prime computers [Farr 77]. The basic interface includes a full-duplex DMA, a full packet buffer in the interface, and relay logic that allows the interface to be bypassed and looped back on itself.

The medium is either coax or fiber optics. The medium data rate is nominally 10 Mbps, but can be reduced to increase the permissible distance between interfaces. A group encoding scheme is used to encode every 4 bits of data into a 5-bit field for transmission; 5 of the other 16 codes are reserved for use as synchronization patterns (e.g., the token).

The host presents a packet to the interface via the appropriate DMA registers. The packet format includes 8-bit destination and source addresses and up to 256 16-bit words of data. Of the 256 addresses, 247 are reserved for use as physical interface addresses, one address is reserved as a broadcast address received by all interfaces, and the remaining addresses are reserved for unspecified diagnostic functions.

When the interface finishes transferring the packet to be transmitted into the interface's packet buffer, it waits for a token. Upon seeing the token, the interface transmits the packet followed by an acknowledgment field and a new token. Interfaces that recognize the packet address set bits in the acknowledgment field that indicate whether the packet was correctly copied. The transmitting interface deletes the packet as it returns. If the acknowledgment field indicates that an interface missed the packet, or if the packet returns corrupted, the interface will initiate an automatic retransmission. Up to seven such retransmissions are attempted before the interface gives up and signals failure to the host.

Toshiba RCB

The Ring Century Bus (RCB) is a high-speed slotted loop designed by the Toshiba research and development center. An initial system based on 10 Mbps links was implemented in 1976, and a 100 Mbps system using fiber optic links was begun in 1978 [Okuda 78].

The basic architecture of the RCB consists of a dual fiber optic loop that links interfaces and a control station. One of the loops is used for data transfer, and the other loop is used by the control station to control data interfaces.

The control station generates the framing pattern on the data loop and monitors the data stream to detect failed interfaces. If the control station detects a failed interface, it issues commands on the control loop that bypass the failed station.

The framing pattern on the data loop consists of 8 slots, each corresponding to a separate TDMA channel. The slot format includes 8 bit source and destination addresses, a 20-byte data field, and several flags. An interface with a message to transmit scans slots on the data loop until it finds a free slot. It then fills the slot with the first 20 bytes of the message. The remaining parts of the message are transmitted 20 bytes at a time over the same channel. Once the entire message has been transmitted, the interface marks the channel as free. First packet, last packet, and acknowledgment bits synchronize the receiving and transmitting interface.

Thus a given interface can never use more than one eighth of the bus bandwidth at a time. After discounting the slot overhead and one-eighth bandwidth factors, the aggregate data rate per channel meets the 8 Mbps design goal for a 100 Mbps loop.

TRW

Researchers at TRW have built a ring network specifically designed for use with a high-speed (more than 100 Mbps) fiber optic medium [Blauman 79]. The following design criteria were used:

1. The network must not have any distinguished nodes that constitute a single point of failure; network growth should be possible simply by adding a new interface to the ring.
2. Address recognition in the interface should include functional addressing as well as physical addressing.
3. Because of the high speed and very low error rate of the medium, error detection and protocol overhead are of minimal importance.
4. The high speed of the medium relative to the computer elements necessitates steps to partition the medium's bandwidth into multiple channels the computer elements can keep up with.
5. The design should minimize and isolate the parts of the interface that must run at line speed because of the higher cost of the required ECL components.

These criteria led to the development of "labeled slot multiplexing," a transmission protocol that combines a fixed-size slot-insertion system, a message-chopping protocol, and prompt acknowledgments into a system which allows for the transfer of arbitrary length messages between hosts.

Each interface is composed of two parts: a high-speed line interface for the slot protocol and a processor interface which handles DMA transfer of messages between the host and the interface.

The heart of the line interface is a shift register which is large enough to hold a complete slot. Slots have the following format:

1. A flag bit. This bit is set in the first and last slots of a message.
2. An ack bit. This bit is cleared before transmission of the slot and set by any interface that copies the slot.
3. An 8-bit field. If flag is set, this field is a destination address. Destination zero signifies end of message. If flag is cleared, this is a data byte.
4. A 6-bit origin address. This is the physical address of the host that transmitted the slot.

The line interface transmits the slot by inserting the shift register in series with the ring. The slot circulates around the ring until it returns into the shift register of the transmitting interface, where it is deleted. The slot is recognized by its origin address. The TRW network periodically discards all data on the ring to insure that damaged packets do not circulate forever.

Interfaces receive slots in one of two ways depending on whether the interface is waiting for the start of a message addressed to it or waiting for the next slot of a partially received message. If the receive side of the interface is waiting for the start of a message, it looks for any slot with a set flag bit (indicating a first slot), and a destination address that is recognized by the host. Except for certain special broadcast addresses, the ack bit must also be off for the slot to be copied.

Addresses are recognized by using the 8-bit destination code as an address into a 256 x 1 bit RAM; the host sets the contents of the RAM to correspond to the addresses it wishes to recognize. By convention, 63 addresses are reserved for physical interface addresses, address zero is reserved for last slot designation, and the remaining 192 addresses are reserved *for function names*. The function name address space is further divided between those addresses that require that the ack bit be cleared and those that do not. The function addresses that don't require a clear ack bit can be used as broadcast channels; potentially several interfaces will copy the message. Those that require a clear ack bit are used to automatically allocate a request message to one of several equivalent servers.

Once the initial slot of a message is copied, the receiving interface remembers the origin address of the initial slot and accepts slots from only that physical address. Thus each message transmission ties up the transmit side of one interface and the corresponding receive side for the duration of the message. Note that the initial slot carries no data; the data bytes arrive, one per slot, in the second and succeeding slots. The end of the message is signalled by a slot from the correct address with a zero data byte and a set flag bit.

One interesting aspect of this design is the lack of any error detection in the message and slot protocols. Software CRCs are used in higher level protocols, but [Blauman 79] suggests that the fiber optic links are sufficiently error free to make this largely superfluous.

IDA

A network design proposed by the Institute for Defense Analysis (IDA) has a slotted system in which slots are circulated in parallel along a 24-bit wide loop. The medium is actually a centralized set of 24 circular shift registers with a latency equal to the number of hosts on the loop. Each interface "owns" one slot, which it periodically reads to receive messages. Interfaces transmit a slot by filling the slot belonging to the desired destination. As the system uses a central clock, no synchronization problems exist, and hence this would have been a very simple and high-performance network.

2.4 COMPARISON AND ASSESSMENT

The interfaces surveyed in this chapter exhibit a great deal of diversity in philosophy and design. Some of the reasons for this are the following:

1. Differences of opinion between researchers.
2. Different system environments and goals.
3. Differences in the available components at different points in time.
4. Different levels of experience and knowledge of prior art.

Nonetheless, it is possible to draw some general conclusions and guidelines relating to the design of local network interfaces; some areas of design even seem to be approaching consensus. This section of the report examines the major interface design decisions and attempts to determine the degree to which each area is an open or a solved problem, and state any applicable tradeoffs, guidelines, or heuristics.

Medium Selection

The three candidates, in order of increasing bandwidth, noise immunity, and cost are twisted pair, coax, and fiber optics. With present technology, coax is probably the only appropriate medium for a bus system; all three can be used for point-to-point connections. Fiber optics is a relatively new technology; many of its drawbacks will be eliminated, and its costs reduced, as fiber manufacturers improve the present state of the art. The strengths of fiber optics are high bandwidth, high noise immunity, and a nonelectrical medium's inherent isolation. It may well be the medium of choice for future point-to-point systems.

Medium Signalling

The two design decisions for the medium are choosing a serial versus a parallel path, and choosing between a baseband and carrier modulation system.

There seem to be very few cases in which a bit serial medium isn't the appropriate choice. Given the existing implementations of coax systems running at 50 Mbps [Thornton 75] and fiber systems at 100 Mbps and beyond [Okuda 78, Rawson 78], speed doesn't seem to be a problem. The exceptions are limited to those systems that have very short line lengths or those systems which must use existing twisted pair.

The choice between baseband and carrier-based transmission is a tradeoff between the additional capabilities of broadband systems versus their cost. Carrier systems can drive longer cable lengths, can use FDMA to provide multiple channels, may be able to share existing CATV media, and can use available CATV repeaters to build very large dual cable busses similar to the Mitre system. Broadband systems can be very cost competitive when they employ standard TV components and have multiple stations to share the cost of the headend repeater; they are less competitive for smaller systems and systems which require custom design.

Medium Topology

The choice here is between bus systems and circular systems. The relative merits of these two systems have long been debated, with the argument centering around the issues of reliability and ease of use.

Reliability

The virtue of a bus system is that the medium may be completely passive, i.e., consisting only of a cable, while circular systems need active repeaters. (Note that the risks of cable failure are somewhat better for a bus system due to less cabling; however, cable lengths are not vastly different because existing bus systems can't support branching.) Either system fails if the cable is cut. Circular systems can reduce the probability of repeater-induced failure by minimizing essential repeater logic and by using automatic bypass relays, but some additional risk remains. Although experience suggests that the additional risk isn't significant [Saltzer 79], it may still be of interest.

Circular systems do enjoy an advantage when reliability is pursued through active measures, i.e., where survivability rather than freedom from failure is important [Zafiropulo 74, Hafner 76, Hopper 79].

A cable break in a circular system can be isolated by the downstream flooding technique used by the Cambridge loop. Locating breaks in cable busses usually involves the use of a time-domain reflectometer (TDR). The TDR, sometimes called a "cable radar," can determine the length of an unterminated cable by measuring the time for a pulse to reflect back from the break. However, TDRs are fairly expensive ($5000), and the technique can't be implemented in a standard interface using standard data transmission as the Cambridge technique is.

Reconfiguration of a broken ring via alternate point-to-point paths is somewhat simpler than generating a new bus.

Locating an interface that fails "on," i.e., fails by generating a continuous level or stream of data, is somewhat simpler in circular systems.

Ease of use

The transmission system in a circular system is somewhat more complex because the interface must be able to repeat transmissions and delete transmissions. The repeater requirement means that some form of phase locking is necessary to prevent bit errors due to clock differences between interfaces. The requirement that interfaces explicitly delete transmissions complicates interface design.

Circular systems using source removal support the design of prompt acknowledgments and other forms of information transfer from the message destination back to the source better than bus systems. Although such facilities can be designed into bus systems, error control is more difficult because the source has less information regarding transmission errors.

Access Control

One measure of the effectiveness of an access-control system is how efficiently it uses the medium. However, efficiency is of limited concern in most local networks because the attached hosts can usually only use 5-10 percent of medium bandwidth.

The efficiency characteristics of a token controlled circular system and an an Ethernet style collision system are actually quite similar [Agrawala 77, Agrawala 78]. Both utilize the medium efficiently, even under heavy loads. In lightly loaded systems, the Ethernet approach has somewhat lower latency, but the difference is usually only a few bit times.

Slotted systems and message-chopping techniques tend to be expensive in terms of medium efficiency. They are primarily useful in situations where the medium data rate is greater than the processing rate of the interface or the data channel to the host, or in situations where it is desirable to be able to simulate FDMA distribution of bandwidth. Some evidence exists to suggest that slotted systems with a central controller can result in simpler, and hence cheaper, interfaces.

Insertion techniques offer enhanced medium throughput through their abilities to overlap transmissions and to queue multiple transmissions in the medium. However, it is less than clear that the increased complexity of the system is a more cost-effective approach than simply using higher bandwidth data links. Another point is that insertion techniques are superior only under heavy load conditions, and few networks offer heavy load.

Host Interface

A persistent problem in the design of local networks is that the cost of implementing a high throughput interface is often repeated for every host type that needs to be supported. Some design time can be saved by using DMA interfaces, etc. produced by manufacturers, but the cost of these components is often disconcertingly high in proportion to the cost of the rest of the interface. The situation is better in the case of recent LSI microprocessors as they often have low-cost DMA chips or chip sets available.

Even if cost is not a problem, the characteristics of the host interface can often be restrictive. For example, the channel protocols in use on large machines often assume that the channel, and not the device, should control the transfer. Thus the interface can't share data structures in host memory.

Interface Buffering

Given that the medium, the interface, and the host are unlikely to have totally compatible data rates, some amount of buffering is essential in the interface. Fortunately, the cost of the necessary FIFOs or RAM buffers has dropped to the point that buffering is easily affordable.

Interface Architecture

The advantages of a programmable architecture seem compelling. However, the speed of existing MOS microprocessors is probably inadequate for high-performance applications. The primary problem isn't the basic instruction rate, but rather the ability to switch rapidly between contexts serving the medium, host status and command registers, and DMA operations. The obvious solution seems to be parallel processing elements in the interface; design work is needed in this area.

Custom LSI is another attractive alternative. However, the high design costs (more than $100K) for a chip of reasonable complexity limit experimentation in this area. The consortium Ethernet is the only *real effort* in this area and may well revolutionize the local network area. However, this device isn't expected to be available until the fourth quarter of 1982 [DEC 80].

Formatting

The message formats for contemporary local networks are fairly similar in terms of the fields provided. However, there seems to be a trend toward providing higher level services in the interface, so as to move more of the high-level protocol mechanisms into the interface. This trend has two natural consequences: more processing power is required in the interface; more fields in the message format are required to provide the necessary control information. Existing examples are the naming facilities of the LNI and consortium Ethernet and the inclusion of prompt acknowledgments in the message format.

3. CONTEMPORARY COMMUNICATION ENVIRONMENTS

3.1 CRITERIA

This chapter studies two existing communication environments: DCS and TCP. The purpose of this study is to estimate the functionality required in the environments' internals and the features required at the user interfaces.

The first problem in trying to study communication environments is that it is often not apparent where the line should be drawn between the application and the communication environment. Most people would regard the ARPANET File Transfer Protocol (FTP) [NIC7104 76] to be a utility rather than an integral part of the communication environment. Yet net mail, the most frequent use of the network, uses FTP as its basic interface to the communication environment. In other systems, such as Cm* [Fuller 78], there is no facility corresponding to FTP because data transfer between machines is incorporated into the processors' instruction set.

Thus, the distinctions used herein are largely philosophical in nature. For the purposes of this study, the benchmark level of communication is the process-to-process level which removes the effects of transient errors (i.e., the data stream is delivered in the same order and has the same content that it had when transmitted.) This level corresponds to the third level in the hierarchy used in [Postel 81a]:

1. Media or local network interface
2. Datagram or internet level
3. Interprocess communication level
4. Applications level

Of course, not all systems have all of these levels or even any layering at all. The DCS software has two layers. Most X.25 systems [CCITT 77] appear to the user as a single layer and hence do not have a datagram level visible by the user, although they may have one internally. The distinction between layers is also blurred when networks are connected; network A may use network B's highest level protocol as a low-level protocol of network A.

Two systems are used as representatives for this chapter: the message system of the Distributed Computer System (DCS) and the Transmission Control Protocol (TCP).

The DCS communication system is chosen because it illustrates a simple, message-oriented protocol designed specifically for a local network-based distributed system. It is also of interest because it uses a fairly intelligent interface designed to support the communications environment through special facilities such as the name table and prompt acknowledgments.

TCP [Postel 81a] is a stream-oriented protocol designed for use in packet-switched computer networks and especially in interconnected systems of such networks. TCP is a DoD standard, and will eventually replace the NCP in the ARPANET [Postel 81d]. The services provided by TCP are similar to those provided by the NCP; the major differences between TCP and NCP are related to the protocol's internals. TCP internals were designed to achieve an orderly and coherent structure and to minimize the assumptions regarding underlying layers. TCP's design assumes that it will usually be supported by the Internet Protocol (IP) [Postel 81b]. IP provides a datagram service, and hence TCP is responsible for reliability measures to recover from lost packets, duplicated packets, etc.

In order to compare these systems we need a means of cataloging their salient features. To describe the communication environment, we need at least the answers to the following questions:

1. Who are the parties to a connection? How are they specified?
2. How is the connection initialized?
3. What kinds of information can be transferred over the connection?
4. How is the connection terminated?

The implementation structures underneath these services also need to be examined:

1. How much "intelligence" is required at each stage of a connection's life to execute the required algorithms?
2. How much state information is required at each level?

The study develops implementation outlines for both protocols. The outlines concentrate on the parts of a protocol server that are fixed by the design of the protocol or are common to any user interface. They exclude the parts of a server that are matched to a particular host interrupt system, process structure, etc. This decision divides the implementation outlines into two sections: simple sequential pseudoprograms that describe protocol processing, and separate discussions regarding the control processing required to multiplex multiple processes and conversations. The discussion develops an agenda of issues to be addressed in later chapters.

3.2 DCS

Overview

The DCS communications environment is a simple protocol for transferring messages between processes. The protocol and ring hardware were designed together to form an efficient unit. The most noteworthy examples of this coordination are the RI's name table, which allows all transmissions to be addressed to process names directly, and the RI's prompt acknowledgments, which are used in lieu of separate acknowledgment transmissions.

Operations

From the user process's point of view, communication consists of

1. The transfer of messages to a name which is either a unique process name or a broadcast name for some set of processes.
2. Reception of the next message on the process's input queue.

Messages consist of a fixed size header which contains source and destination process names, and a variable length data section. Any process may send a message to any name at any time; no initial connect or closing protocol is visible to the process.

User calls on the communication environment are shown in Table 3-1.

Table 3-1: DCS communication environment user calls

```
Send(Name,Address,Length)

Ctrl(Name,Address,Length)

Recv(Address,Flags,Timeout)
```

`Send` transmits a message to the process or broadcast name specified by the first argument. The data to be transmitted and its length are specified by the second and third arguments. The message is delivered to its destination(s) before control is returned to the process that called `Send`. Delivery means that the message is placed on the destination's input queue, not necessarily read by the destination process. `Send` returns a value that corresponds to a composite **match** and **accept** value for all transmissions. The transmitting process uses this value to determine whether message transmission was successful.

`Ctrl`, like `Send`, transmits a message. However, the message is delivered to the process which controls the process named in the first argument, rather than the process itself. Control messages are usually interpreted by the operating system and are used to control process execution.

`Recv` allows a process to wait for and receive a message. The `Flags` argument specifies the conditions that cause completion of the `Recv` call; the process can wait for a message, a timeout, or whichever of the two occurs first. The `Address` argument specifies a buffer in process space that is to receive the message. The first word of the buffer contains the length of the buffer; if the arriving message is too long, it is truncated. The message is copied following the length word. If the `Address` argument is zero, the `Recv` call doesn't copy the message; it only sees if one is available. `Recv` returns a value of true if the call terminates due to the message condition, false otherwise. `Timeout` is the maximum amount of time the `Recv` call can block waiting for a message.

Message Format

The internal format for messages is shown in Figure 3-1. The fields are the same as those recognized by the RI, with the exception of the added 16-bit word following the message-length field. The RI regards this word as part of the message data; the software in the hosts treats it as part of the header. This word is removed before the message is passed to the receiving process.

The Message Definition Field (`MDF`) defines the type of the message. There are four types: normal, control, purge, and purge request. Normal and control messages are generated by processes; purge and purge request messages are internal to the communications system and are not seen by the user. The use of these messages is described in the next section.

The SOM and EOM bits are used to flag packets that are the first and last packets of a potentially multipacket message. Purge and purge request messages are short enough to never require packeting; hence only messages sent directly by the user (normal and control) need these fields. In a single-packet message both bits are set.

```
                                1
0              7                5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              DPN              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              OPN              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| | |S|E| | | | |               |
| | |O|O| | | |S|      MDF      |
| | |M|M| | | |B|               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |
/             Data              /
|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3-1: DCS internal message format

The SB bit is the sequence bit for this packet. The sequence bit is meaningful for data and control messages and is used to drive an alternating-bit protocol [Lynch 68] for duplicate message detection. Purge and purge request messages aren't sequenced so they can be used without relying on the existence of synchronized sequence spaces. Their semantics are defined so that duplicates don't cause problems.

Connection State Machine

Although connections aren't seen by the user, they are very much a part of the communications system. A connection is a simplex path between a unique source name and a unique or broadcast destination name. A connection exists if matched connection records exist at the source and destination(s).

In the DCS implementation, each process has eight transmit-connection records and eight receive-connection records in its process control block. The communication software uses the process's eight receive- and eight transmit-connection records as working sets: Whenever a message is sent to or received from a new name and no connection record is available, the least recently used connection is closed and its connection record reclaimed. The communication protocol uses the purge and purge request messages to coordinate connection record use at both ends of the connection.

Transmit-connection records include the following:

1. The destination name
2. The next 1-bit sequence value
3. A lock bit
4. An 8-bit LRU field

Receive-connection records include the following:

1. The destination name
2. The origin name
3. The next 1-bit sequence value
4. A lock bit
5. An 8-bit LRU field

The records are asymmetrical because messages always have unique names in the origin field while the destination field may be either a broadcast name or a unique name. The transmit-side connection record doesn't need an origin field because the connection record is associated with the process and hence with its unique process name. The receive side needs a destination field because messages may be addressed to a broadcast name of the process rather than to the unique process name.

The connection-management strategy used depends heavily on the fact that only one transmission can be in progress at a time on a connection. The DCS connection-state machine for a sender is shown in Figure 3-2; the receiver needs no state other than knowing whether a connection record exists or not. This figure shows processing associated with normal connection processing; transitions and states dealing with error conditions aren't shown.

In the following discussion of the state machine, nonbroadcast messages are assumed, although the operation for broadcast transmissions is often identical. The differences are discussed at the end of this section.

<u>Opening a connection</u>

The Closed state isn't really a state at all, but rather a condition in which the source has no state information regarding the specified destination. The same condition should be true at the destination(s), although this need not be the case. The protocol is designed so that unilateral loss of connection-state information doesn't interfere with proper operation. A source that loses its state can transmit new messages successfully. A destination that loses its state will receive new messages correctly. Note that the protocol doesn't deal with the protection of old messages or with detecting host restarts.

When a process sends a message to a destination for which there is no local transmit-side connection state, the protocol first sends a purge message to the specified destination. The purge message isn't sequenced and hence will always be processed if received. The purge message instructs the receiving communications system to discard existing connection-state information. Thus if the transmitting host crashes and restarts, old connection-state information in the receiving host won't be reused. The purge transmission corresponds to the arc between Closed and Initialize_Lock.

If the purge message is not recognized by any of the RIs, then the addressed process doesn't exist. This condition is signalled by a **no match, no accept** transmit status in the returning prompt ack. In this case the transmitting process is immediately informed that the destination doesn't exist; there is no point in transmitting the actual message. The connection record is discarded and the connection returns to Closed state.

```
                    +------------------+
                    |      Closed      |
                    +------------------+
                              |
                              | Send / Xmit purge
                              V
                    +------------------+
/------------------>|  Initialize_Lock |
|                   +------------------+
|                    |        |
\-------------------V         | No match,/ Create entry,
  Match / Rexmit              V accept     Xmit message
                    +------------------+
/------------------>|   Message_Lock   |
|                   +------------------+
|                    |        |
|<------------------V         | No match,/ Toggle sequence
|  Match / Rexmit             V accept
|                   +------------------+
<-------------------|       Idle       |------------------>|
   Send / Xmit      +------------------+                   |
     message                  |                            |
                              | Purge request / Xmit       |
                              V                 purge      |
                    +------------------+                   |
/------------------>|    Purge_Lock    |<------------------V
|  Match / Rexmit   +------------------+       LRU death /
|          purge     |        |                Xmit purge
<-------------------V         | No match /
                              V Delete entry
                    +------------------+
                    |      Closed      |
                    +------------------+
```

Figure 3-2: DCS connection-state machine

Match causes retransmissions until **no match, accept** status is returned, or until a designated number of retries are completed. **Match** conditions are usually caused by a host which does not reenable its RI input quickly enough after a prior message. Hence retransmission after an timeout is the appropriate response. The number of retransmissions should be limited because a long term **match** condition can be caused by a host crash or a severe shortage of receive buffers.

Transmitting data

As soon as **no match, accept** status is returned for the purge, the actual message text is transmitted. The first message transmission corresponds to the arc between Initialize_Lock and Message_Lock. When this message arrives, no connection state will be present at the receiving host, so a state record is created using the sequence value in the arriving message. The connection stays in the Message_Lock state until the message gets through (**no match, accept**) or the retransmission mechanism gives up (**match**). The connection stays in the Idle state between message transmission requests and loops back to Message_Lock for each new message transmission.

The sequence bit in the transmitting connection control record is toggled when any message is successfully transmitted so that the next transmission will have a different sequence value. The receiver toggles the value of the sequence field in its connection-state record whenever a new message passes the sequence test. Because the receiving process discards transmissions that arrive with sequences opposite to the value in the receiving-connection state, duplicates are suppressed.

A received message which passes the sequence test is used to satisfy a pending `Recv` call, or if no `Recv` is pending, the message is placed on the input message queue. Control messages are sequenced using the target process's connection record, but delivered to the target process's superior.

Connection termination

The connection persists until one of the ends decides to terminate it. A connection can be directly purged by the transmitting end; the receiving end uses an indirect method. Either end terminates the connection if it needs to reclaim the connection record's memory, or if the associated process is to be terminated. If the transmitting end decides to terminate the connection, it does so by sending a purge message. If the receiving end wishes to terminate the connection, it does so by sending a purge request to the transmitting end, which sends back a purge message. The request strategy is used rather than having a purge transmitted directly by a receiver to avoid synchronization problems should the transmitting end be transmitting a message at the same time a purge is issued by the receiver.

Broadcast

When processes are created, the kernel loads the RI's name table with the unique name of the process. If the process belongs to a broadcast group (such as all file handlers), the kernel also loads the broadcast name(s) for the process into the RI's name table. The same broadcast name is also loaded into other RIs by the other members of the broadcast group. Thus multiple RIs can have the same name in their name tables and will attempt to receive messages addressed to the broadcast name.

Broadcast transmissions can be managed by the same algorithms except in the case of a prompt ack containing both match and accept. A **match, accept** response to a broadcast transmission means that some receivers have updated sequence states and some do not. Retransmissions may cure this problem, but the transmitter can't be sure unless it gets the unambiguous **no match, accept** response; if the transmitter gets consecutive **match, accept** conditions, it cannot be sure if the same destinations are missing the message every time or not.

The policy followed by the DCS system is to attempt to get a pure **accept** response for some number (currently 16) of retries and to update the sequence space following the last retry. Thus members of the destination set that miss all 16 retries of one message will necessarily miss another message before the sequence spaces resynchronize. This allows for a broadcast set to continue operation even though some of the destinations are in dead hosts.

3.3 DCS IMPLEMENTATION OUTLINE

The implementation of the protocol is divided into separate code sections for the support of the receiving and transmitting primitives and communication data structures associated with each process.

Transmit Side

Figures 3-3 and 3-4 are pseudocode for the DCS Send function.

The top-level code shown in Figure 3-3 corresponds very closely to the actual code which is linked into the process context. The subprocedures in Figure 3-4 illustrate the processing that takes place at a lower level, but these procedures don't model control flow accurately. In the real DCS, the lower level procedures are initiated by an operating system call from the Send procedure, and loop iterations in either of these procedures are driven by RI interrupts and timer interrupts. In the model all of the interactions between the host and the interface for a single transmission are modelled by the call to RIsend. These models also ignore the multiplexing that takes place between different processes on the same host.

All three procedures return a three-bit value, with the three bits corresponding to the **accept**, **match**, and error conditions. In the DCS system the error bit is always handled by the system and never seen by the user. Functions are divided among these procedures as follows:

- The Send procedure is responsible for reserving a connection record and associating it with each call to Send, breaking long messages into packets, updating the sequence bit in the connection record whenever a transmission succeeds, and composing the appropriate return code based on the response to transmissions. If the message fits in a single packet, the Send return code is simply the value of the response bits returned by Send_pkt for the packet; if multiple packets are required, the returned value is the "worst" value for any single packet. If any packet of a multipacket message can't be sent, or if the attempt to set up the connection fails, the remaining packets are omitted and the appropriate return code is immediately returned.
- The Send_pkt procedure controls the transmission and retransmission of a packet to compensate for temporary **match** conditions. Match conditions are usually caused by the delay in enabling the network interface following a reception; hence the timeout should approximate this delay. A crashed host at the destination will never enable its input, so retransmissions should be limited.
- The Send_once procedure controls the transmission and retransmission of packets to compensate for transmission errors. In general this type of error is not related to host conditions at the message source or destination, so the retransmission timeout and retry count should be selected to match the access-control protocol of the medium. For example, using the LNI system, there is no reason for any delay, whereas a system using a contention medium would use this timeout to compensate for congestion on the medium.

```
procedure Send(Name,Address,Length);

   {Find or create a connection record}
   lookup_connection(Origin,Name);
   if not found
   then begin {if no connection, create one}
        status:=send_pkt(Origin,Name,purge,purge_msg,0);
        if accept in status
        then create_Conn_record {purge successful, set up connection}
        else return(status) {if purge fails, so would data}
        end;

   lock_Conn_record; {prevent other use of this path}

   {Send all the packets of the message}
   status:=[];        {initialize status for whole message}
   while Length>0     {break data into packets if necessary}
        begin
        send_length:=min(Length,max_packet_length);
        {copy sequence value from conn record into packet}
        tstatus:=Send_pkt(Origin,Name,normal+SB,Address,send_length);
        if accept in tstatus
        then begin
             {merge status for this and previous packets}
             status:=status+tstatus;
             toggle_SB; {toggle transmitting conn record SB}
             Address:=Address+send_length; {setup next packet}
             Length:=Length-send_length
             end
        else begin
             unlock_Conn_record;
             return(tstatus) {return worst status}
             end
        end;

   unlock_Conn_record;

   return(status);

end; {Send}
```

Figure 3-3: DCS Send top-level code

Transmit-Side Enhancements

The RI and LNI support a level of service similar to that modeled by the RIsend procedure. If the DCS transmission protocol is to be supported more fully by the interface, the outline suggests five possible levels of support. These levels are described below:

1. Implement transmission error retry in the interface (Send_once)
2. Implement transmission error and match retries in the interface (Send__pkt & Send_once)

```
procedure Send_pkt(From,To,Type,Address,Length);

   {this procedure retransmits a packet trying to avoid match}

   status:=[];
   for i:=1 to max_match_retry
        begin
        tstatus:=Send_once(From,To,Type,Address,Length)
        if (tstatus=[accept]) | (tstatus=[])
        then return(tstatus)
        else begin
             if accept in tstatus then status:=[accept];
             wait(match_timeout)
             end;

        end;
   return(status+tstatus)

end; {Send_pkt}

procedure Send_once(From,To,Type,Address,Length)

   {This procedure transmits a packet until it is transmitted
    without transmission error, or until maximum retries exceeded}

    for i=1 to max_err_retry
    do  begin
        status:=RIsend(From,To,Type,Address,Length);
        if not(err in status)
        then return(status)
        else wait(error_timeout)
        end;

   return(status)

end; {Send_once}
```

Figure 3-4: DCS Send subprocedures

3. Implement packeting and both retry mechanisms in the interface (while loop from Send, Send_pkt & Send_once)
4. Implement connection locking, packeting, and both forms of retry (Send, Send_pkt & Send_once)
5. Implement connection record management (creation and deletion), connection locking, packeting, and both forms of retry (Send, Send_pkt & Send_once)

The algorithms for any of these levels are simple if the interface processes one Send request at a time. If multiple requests are queued in the interface, then a multiplexing policy must be designed.

The need for multiplexing and the complexity of the required control structure are a function of the level of service provided by the interface and the expected performance characteristics of the medium. For example,

- Transmission errors aren't related to a packet's contents, and level-one timeouts are between interface accesses to the medium, rather than being related to a specific packet. Hence, a level-one multiplexing interface could serve requests to completion before considering the next queued request; there is no point in considering more complicated schemes.
- The situation is different for **match** retries included in level two. Here the timeout is related to the packet arrival service time at the destination, and hence the interface should attempt to serve a Send request to a different destination whenever a match retry is needed. Timeouts for different destinations can run concurrently. This strategy avoids holding up all requests due to congestion at one destination.
- If the host is incapable of generating multiple requests or if it generates requests much more slowly than they can be serviced, no multiplexing policy can be justified.

These considerations, based on the existing system, combined with opportunities for improvement in services, lead to a functional list of areas that need study:

1. More powerful addressing facilities, particularly for broadcast
2. Management policies and accessing primitives for the connection record database
3. Request queueing, selection and service--perhaps using the connection record database as a repository
4. Sequencing management
5. Methods for decreasing the overhead implied by the processing of a single Send request

Receive Side

Message reception involves three types of activities:

1. Creation and deletion of destinations
2. Processing of arriving packets
3. Satisfying user Recv calls

The creation and deletion of destinations, and hence the associated name table entries, are infrequent activities that usually correspond to the creation and deletion of processes. These services are simple to provide because they can be performed in a purely synchronous manner, i.e., the interface can change the name entry while the host waits.

The main task consists of establishing a correspondence between incoming packets, which arrive unpredictably on the medium, and Recv requests, which arrive unpredictably from the user. In general, it is impossible to depend on the ordering of the two events or the amount of time that will elapse after the first event has happened until its corresponding partner event happens.

Figure 3-5 is pseudocode for the processing that takes place when a packet arrives.

Incoming packets (represented by argument "p") are processed using data in the appropriate connection record (represented by the structure "cr"). Processing is divided into three routines: one routine for each of the communication system messages (purge and purge request), and one common routine for user generated message types (data and control).

```
procedure Packet_arrives(p:packet);

{Separate processing by message MDF}
case packet.MDF of

purge:begin
      lookup_connection(p.OPN,p.DPN);
      if found
      then delete_Conn_record
      end;

purge_request:
      begin
      lookup_connection(p.DPN,p.data);
      if found
      then if not locked {see if connection record is locked}
           then begin
                lock_connection;
                send_purge; {send the requested purge}
                delete_Conn_record
                end
           else queue_purge;
      else send_purge {respond to a purge for a non-existent connection}
      end;

data,
ctrl: begin
      lookup_connection(p.OPN,p.DPN)
      if not found
      then begin
           create_Conn_record;
           cr.SB:=p.SB
           end;
      if p.SB=cr.SB {verify that sequence in packet is expected one}
      then begin
           toggle_SB; {change cr.SB}
           queue_packet
           end
      else discard_packet {packet is a duplicate so discard it}
      end;

end; {case}

end; {packet_arrives}
```

Figure 3-5: DCS packet arrival processing

Purge message processing

Purge messages originate in the send side of connection management and are processed by the receive side. The originating send side generates the purge message in response to the termination of the associated sending process, or if the send side needs to reclaim the connection-record resource. Connection locking is under control of the transmit side; a receiver must always honor a purge message. The receive side can expect to receive purge messages for nonexistent connections due to the precautionary purge sent when the connection is created, or as the result of host crashes or duplicates for broadcast purges.

Purge-request message processing

Purge-request messages originate in the receive side of connection management. Like purge messages, they are caused by a need to reclaim receive connection records, or as a result of process termination.

The connection is controlled by the send side. The purge-request message acts as a prompt for a purge message and has no direct effect on connection management itself. The receive side constructs the purge-request message and sends it to the host which manages the other end of the connection. Normally a purge message will be sent in response, although the purge request can be ignored if the connection is locked.

Purge-request messages are more complex to process then purge messages because they interact with send-side connection management and because they require access to a possibly locked send-connection record. Two policies are available to deal with the locked resource problem:

1. The purge request can be discarded if it cannot be immediately processed.
2. The purge request can be queued until it can be processed.

These choices generate different divisions of responsibility between the protocol mechanism that generates purge-request messages and the protocol mechanism that consumes the purge-request messages. The discarding policy places more responsibility on the originator; the queueing policy places more responsibility on the consumer.

If the consumer discards purge requests that can't be immediately completed, then the originator must retry the request. One advantage of this strategy is that the originator knows whether the purge request is to free resources or to close a specific connection, and hence can use different strategies for the two cases. If freeing resources is the goal, then the originator can attempt several purge requests in series or parallel. If the goal is to clear a specific connection, then the same purge request should be retried. A second advantage of this strategy is that the retry policy will compensate for failures in the request_purge consumer.

Purge requests can be queued in the consumer while the transmit-connection record is locked. Once the connection is unlocked, a purge message is sent back to the purge request's originator. In the DCS system, this is accomplished using system process control primitives and an ancillary sequence bit process. A simpler system could be constructed by incorporating a "purge needed" bit in the transmit-connection record. When a locked-connection record was unlocked, a set "purge needed" bit would cause the generation of a purge message. The purge might also be piggybacked on the last packet before unlock.

Data and control message processing

Data and control messages are sequenced to protect user data; hence they require a receive-connection record for processing. The first part of processing looks up an existing connection record or creates a new one.

If a new connection record is to be created, the system may need to purge an existing receive connection record to free up a record slot. In addition to the possible problems already described with the purge-request processing, the packet must be queued until it can be processed.

Once the connection record is available, the sequence bit in the packet is checked against the sequence bit in the connection record. If the sequence test succeeds, the packet is delivered to the user. The delivery is represented by the `Queue_packet` call. `Queue_packet` has two functions: it adds the new packet to the process's input queue, and it checks for pending `Recv` requests.

A simple way to implement the input message queue is as a linked list of messages, where each message is itself potentially a linked list of packets. This is the DCS strategy. If a new packet has its start of message (SOM) bit set, it is chained onto the end of the message-linked list. Packets without a SOM bit must be chained onto the appropriate message's packet list.

If the packet completes a message, or is in itself a complete message, then the `Queue_packet` procedure must check to see if the new message satisfies a previously queued `Recv` user call. This check can be triggered by the EOM bit in the packet and must interface with the operating system's process-scheduling primitives.

Satisfying user Recv requests

The processing steps described so far create an input queue of messages for each process. The mechanisms associated with servicing `Recv` requests complete the communication process by delivering the message data to the user.

Delivery is possible when a complete message is present on the input queue and a `Recv` request from the process is pending. Because these two preconditions can occur in either order, it is necessary to check for the rendezvous each time a message is completed or the user calls `Recv`. Pending `Recv` requests may be withdrawn due to the expiration of the `Recv` timeout, or by the kernel, when a process is deleted, etc.

In the DCS system, the kernel checks the input message queue when the user calls `Recv`. If a complete message is on the queue, the `Recv` request is satisfied immediately. The `Recv` request may also complete immediately if a zero timeout is specified. If the `Recv` can't be completed immediately, the kernel creates an entry in its timer queue corresponding to the timeout and blocks the process as "waiting for message or time event." Whichever event occurs first causes the process to return to the "runnable" state.

If a rendezvous of a complete message and a `Recv` request occurs, the packets of the message are copied into the buffer specified in the `Recv` request. Once the copy is complete, the packet buffers are returned to the kernel's free buffer pool. The copy procedure has the obvious disadvantage of consuming CPU time. The advantages include hiding the packeting mechanism from the user process and not depending on reasonable packet-buffer management by the user process. Certain trusted system processes bypass the copy procedure and work directly with packet buffers, although most system processes do not.

Receive-Side Enhancements

The design of the receive side of the interface must complement the transmit side and is constrained by cost and performance criteria inherited from the application. Ignoring the cost constraint, the DCS experience suggests the following ideas:

1. The receive side should minimize retransmissions and end-to-end communication delay by accepting packets from the medium whenever possible. The receive side should be full duplex, i.e., it should not be disabled by transmit-side activity. The receive side should be disabled for as short a period as possible following a reception. Methods include automatic receive-side restart following packet arrival and transmission errors. Ideally, the interface should be able to deliver multiple packets to the host without requiring host intervention and the corresponding delay.
2. Because message processing represents a drain on the host's resources (and possibly the interface's), the interface should suppress packet processing that serves no useful purpose, In present systems, this includes CRC error detection and address recognition in the interface. The filtering concept can be extended to include filtering on the basis of sequencing, flow control, security, and similar criteria.
3. The multicast facilities of the LNI system allow for low-cost multicast services. However, they don't perform as efficiently or reliably as possible in the presence of errors or receiver dead time. The expected number of transmissions required to achieve a pure accept grows exponentially with the number of receivers. Improvements should be sought.
4. Several packet-processing steps require access to the corresponding connection record. If these activities are moved into the interface, the interface needs to be able to store and access this data. If the data is used for filtering, the access must be rapid. The speed requirement is related to the minimal packet time on the medium, the amount of buffering in the interface, and the host-interface performance characteristics.
5. Connection m[illegible]ement (in the sense of the purge and purge-request protocol) and Recv request [illegible]agement represent services that might be moved into the interface. In both cases, a definition of services is required. The interface should allow for flexibility in the host's process control and buffer strategies.
6. When connection records are incorporated into the interface, they represent a resource which must be managed. Resource allocation and control mechanisms must be created.
7. The DCS style of communication is based on an environment where resource control isn't a critical issue. For example, connections can be created at the whim of the sender, and the match/accept protocol implies that a destination isn't free to reject offered packets. While this creates conceptual simplicity, it may not be realistic for all systems. The need for flow control in the context of the match/accept protocol is one worthwhile goal.

3.4 TCP

TCP Overview

The present structure of TCP [Postel 81a] is the result of an evolutionary process that had as its basis the ARPANET experience. Using ideas from the ARPANET NCP [Carr 70], TCP evolved to fit a new environment which consists of an interconnection of multiple networks including the ARPANET, various local networks, and networks based on satellite [Abramson 73a,Yuill 76] and packet radio [Burchfield 75, Fralick 75, Frank 75, Frank 76, Kahn 75]. At the same time the designers of TCP took the opportunity to simplify and extend several protocol mechanisms.

The target environment of TCP is the Internet environment. The Internet system consists of many networks, including the ARPANET and several local networks. The networks are connected by gateways. Gateways are hosts that are physically connected to two or more networks. In some cases gateways are normal hosts, and in other cases the gateways are small, presumably less intelligent, hosts dedicated to the gateway task. All of the member networks have, and are expected to continue to have, locally defined protocols of various types and different types of basic communication systems.

The Internet environment must work with diverse resources. Two example members of the Internet system illustrate this diversity: ARPANET and the SATNET. The ARPANET has a basic communication service (the HOST-HOST protocol) that has low latency (less than 100 ms) that varies depending on the number of hops between source and destination, very reliable delivery (error rate of approximately 10**-12), a 50-Kbit bandwidth, and a very distributed, store-and-forward architecture. The SATNET system uses a satellite channel shared by several hosts by means of special interface machines (SIMPs or Satellite IMPs). SATNET communication services have a high minimum latency due to the propagation time to and from the satellite (250 ms), high bandwidth (1-3 Mbps), medium-to-low message reliability (10**-7 to 10**-5), and a centralized node (the satellite). These two systems illustrate the large disparity possible in the Internet system.

Another source of diversity in the Internet system is the different applications that will eventually cross network boundaries. Internet file transfer, terminal sessions, voice, and interprocess communication are examples of applications that place different demands on the communication system. File transfer requires high bandwidth and reliability, but is insensitive to latency. Voice transfers require low latency and high bandwidth, and can't afford the delay inherent in retransmission-based schemes for enhancing reliability. Fortunately, voice systems will tolerate a moderate error rate.

These facts suggested the following design approaches:

- Diverse systems with very different protocol conventions and communication facilities already exist; new networks are expected. Hence, we should adopt a lowest level standard that presumes as little as possible about the member networks and will allow the addition of still different new networks.
- Different applications will use the Internet system, and they need different services. Hence, this lowest level standard should retain the performance characteristics of the network on which it resides and not create virtual levels of service that are less powerful.
- We need standard protocols for certain services. In order to avoid duplication, these

protocols should use the standard bottom layer. In order to allow for generic classes of protocol, the lowest layer should understand this partitioning of the protocol space, but not necessarily every generic type of protocol.

These ideas led to the current Internet system.

The lowest level standard protocol is the Internet Protocol (IP). TCP is the most common second layer protocol. The IP server handles the unreliable delivery of packets (or "segments" as they are known in Internet jargon) for its users. The IP server accepts packets from second-level servers, transmits the packets through as many networks as required, and delivers the packets to the specified destination servers. The TCP layer multiplexes all TCP connections for all processes on the attached host and manages the retransmissions and acknowledgments necessary to do this reliably.

The resulting protocol structure is shown in Figure 3-6.

```
+------+ +-----+ +-----+       +-----+
|Telnet| | FTP | |Voice|  ...  |     |  Application Level
+------+ +-----+ +-----+       +-----+
    |  |           |             |
   +-----+       +-----+       +-----+
   | TCP |       | RTP |  ...  |     |  Host Level
   +-----+       +-----+       +-----+
      |             |             |
   +-------------------------------+
   |       Internet Protocol       |  Gateway Level
   +-------------------------------+
                 |
     +---------------------------+
     |   Local Network Protocol  |    Network Level
     +---------------------------+
                 |
```

Figure 3-6: Internet environment protocol layering

In addition to features required by the Internet architecture, TCP has some protocol simplifications as compared to NCP.

In the NCP, connections are made between "sockets" and are simplex. The socket is very similar to the familiar operating system convention of a logical I/O stream (e.g., TENEX JFN, OS/360 DDNAME). The socket ID is a concatenation of host number, process ID, and a field to allow a process to use multiple sockets. Processes using the network will usually create complementary pairs of connections to achieve full-duplex capability. Odd and even socket numbers are used for output and input connections, respectively.

TCP uses a single socket ID for each end of the connection and makes all connections full duplex. That is, TCP defines a connection to be a full-duplex path between two unique sockets. A short ID, called a handle, is used within a restricted context to shorten the field length necessary to identify a connection.

This change has no effect on the functionality derivable from the message system in that one can always simply not use part of the capabilities of a connection. In practice, most connections between processes need full duplex capability, so this change just serves to simplify things. It is somewhat difficult to argue that a connection is ever really simplex, because a reverse flow of control information is required to insure reliable delivery of information. While this convention has no adverse effect on the complexity of software needed to implement the protocol on a large host, it may affect the structure of an outboard communication environment in that information must be shared between the input and output functions.

TCP Operations

Although the details of the TCP-process interface vary widely from implementation to implementation, some of the operations are defined by TCP itself. Most TCP-process interactions are explicitly initiated by the user in the form of a subroutine or supervisor call to TCP. These calls transfer data and specify control functions.

In addition to user-initiated interactions, TCP should have some way to asynchronously interrupt the user. These asynchronous signals either can be abnormal events (e.g., the host at the other end of the connection has crashed) or are events that the user has indicated should cause such a pseudointerrupt (e.g., let me know when new data arrives).

Combinations of these two types of interaction are common. For example, most TCP implementations allow the user to queue read requests. When a read request is issued and there is no buffered data waiting to be read, the parameters of the read request are queued and control is returned to the user. When the requested data arrives, it is placed in the specified buffer and the user is informed by a pseudointerrupt.

Many variations are possible here, depending on the sophistication of the process environment and the particular TCP environment. The basic user-initiated calls are listed below in Table 3-2. Optional arguments are coded in "[]"s.

The detailed semantics of these calls are discussed in later sections. Briefly, the `Open` call initializes one end of the connection and returns a handle (typically a small integer) that the user can use to refer to the connection in subsequent calls. The `Send` and `Receive` calls are used to send and receive data. `Close` is used to signal that this end of the connection has no more data to transmit and wishes to end the connection. The `Status` call returns information which describes the state of the connection. `Abort` is used to signal a catastrophic error condition meant to result in the unilateral close of the connection.

The TCP server should be able to signal the following events to the user in an asynchronous manner, as shown in Table 3-3.

The `Queue_completion` pseudointerrupt signals the completion of a previously queued request. For example, a call on `Open` might simply start the initial connection protocol while allowing the user process to continue execution. The process would then be notified via a pseudointerrupt when the connection actually was established. Because multiple calls could be queued, some form of request_ID is necessary. The form of this ID and the mechanisms for process blocking, etc., are highly operating-systems dependent.

Table 3-2: User calls to TCP

```
    Open(local_port
        ,foreign_socket
        ,active/passive
        [,buffer_size]
        [,timeout]
        ) returns connection_handle:handle

    Send(connection_handle,
        ,buffer_address
        ,byte_count
        ,push_flag
        ,Urgent_flag
        [,timeout]
        )

 Receive(connection_handle,
        ,buffer_address
        ,byte_count
        ,push_flag
        [,timeout]
        )

   Close(connection_handle)

  Status(connection_handle)

   Abort(connection_handle)
```

Table 3-3: Events signalled by TCP to the user

```
Queue_Completion(request_ID)

Urgent(handle,offset)

Illegal_SYN(handle)

Connection_failure(handle)

Connection_reset(handle)
```

An `Urgent` event occurs when the local TCP receives notification from the other end of the connection that urgent data exists in the receive byte stream beyond the data that has already been read by the local process. This signal is used by one end of the connection to recommend to the other end of the connection that data in the receive byte stream be speedily read to the specified point in the data stream. The exact semantics of this facility are up to the processes using the connection, although its usual function is to signal a break condition similar to that generated by a terminal user.

The last three events are used to signal fatal errors. `Connection_reset` signals that the other end of the connection has aborted the connection. `Illegal_SYN` signals that the local TCP has aborted the connection due to the arrival of an illegal segment. `Connection_failure` signals that the connection aborted because of a timeout.

TCP State Diagram

The TCP state diagram in Figure 3-7 is taken from [Postel 81a]. It illustrates the major states and transitions, but addresses neither error conditions nor actions which are not connected with state changes. In particular, retransmissions resulting from segment loss, abort conditions, and timeouts that cause connection aborts are not shown. Note that both ends of the connection have their own, and potentially different, state variables.

The state diagram may be partitioned into four major sections which correspond to the four stages in the connection's life: nonexistence, "birth," "life," and "death."

The first and last states in Figure 3-7 are labelled "CLOSED." These "states" are not really states at all, but are a notational device to signify that the connection doesn't exist. In certain TCP implementations, this state might be meaningful, but it usually corresponds to a total lack of information about this connection.

The LISTEN, SYN Sent, and SYN RCVD states are used during the birth, or initial connect, portions of the protocol. The LISTEN state is entered by a passive `Open` command and means that the connection is waiting for another process to request a connection. The SYN Sent and SYN RCVD states are entered when the two parties to the connection are in the process of trying to set it up. Data transfer operations cannot be completed during this phase because the connection is not yet established. Some implementations of TCP will allow data transfers to begin during these states, but the data can only be buffered, not delivered.

The ESTABLISHED state of the connection is the main state. Once this state is reached, initial connect is complete and data transfer can begin.

The FIN WAIT, FIN WAIT-1, FIN WAIT-2, TIME WAIT, CLOSE WAIT, and CLOSING states are used while the connection is in the process of being closed. Because both ends of the connection must issue a Close call before the connection will be deleted, it is possible for the connection to persist in some of these states indefinitely.

TCP Segment Format

The format used by TCP segments is shown in Figure 3-8. The source and destination host addresses are carried in the IP header, which precedes and encloses the TCP segment. The Internet header is shown in Figure 3-9.

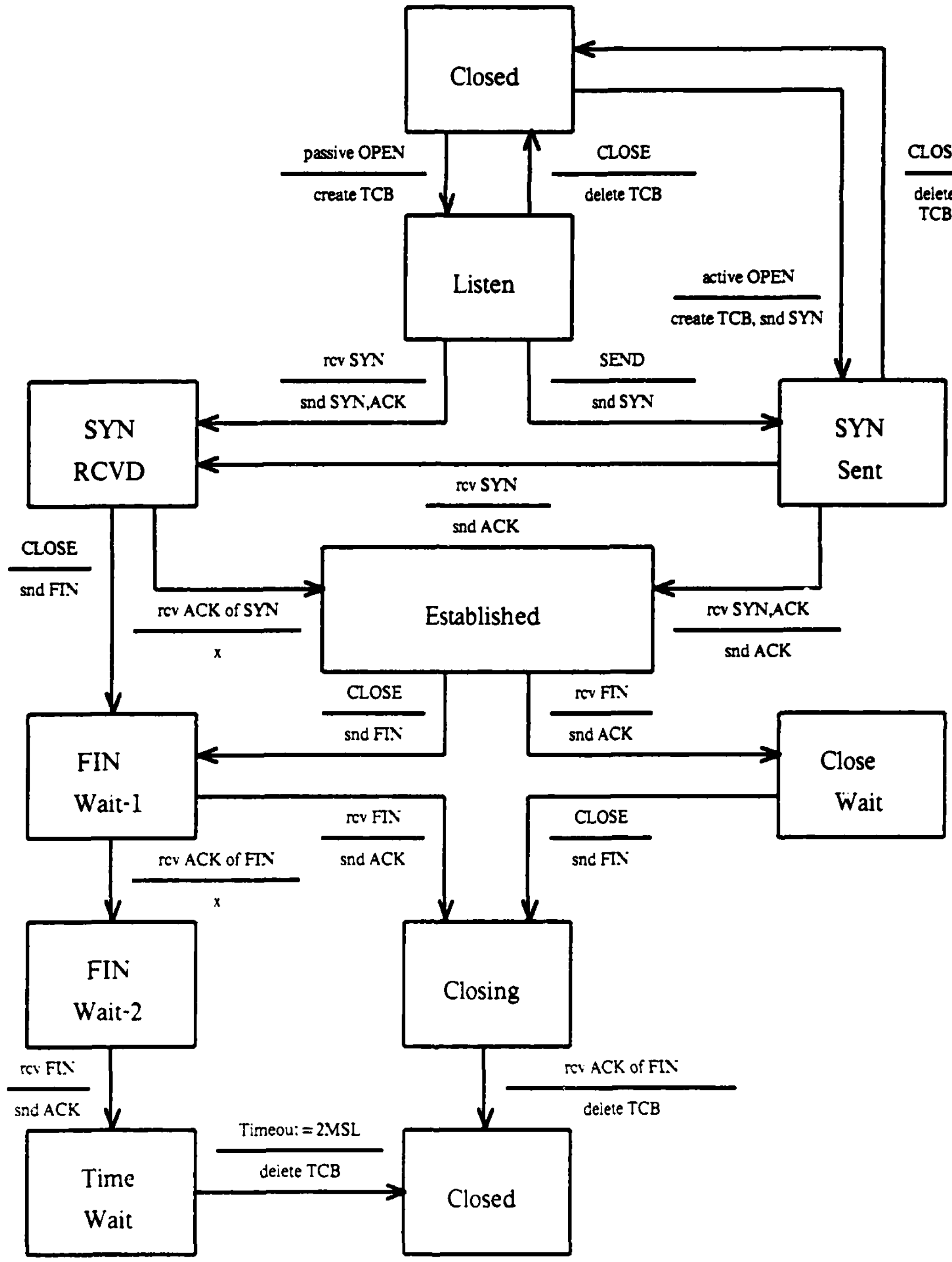


Figure 3-7: TCP connection state diagram

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3-8: TCP header format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3-9: Internet header format

The Internet Protocol Interface

The IP header contains several items which the TCP server finds essential. The total length field in the IP header, less the IP header length (IHL), is the length of the TCP segment. The source and destination host addresses are also in the IP header. Rather than restrict the TCP server to run under IP, the TCP and IP servers pass pseudoheaders with TCP segments when packets are passed between the two servers. The pseudoheader format is shown in Figure 3-10.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Source Address                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Destination Address                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     zero      |     PTCL      |             TCP Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3-10: IP pseudoheader format

TCP Units of Data Transfer

The unit of data transfer in TCP is the octet or byte of eight bits. (The terms byte and octet are used interchangeably here.) Over the life of a connection, the full duplex data flow consists of two (one for each direction) ordered byte streams. The communicating processes can supply or consume data from this stream in any quantity of bytes, but efficiency and buffering considerations dictate that the TCP server remain free to either break up or combine data transfers into segments.

The need to reliably deliver the offered data stream at the other end of the connection, while at the same time allowing maximum freedom in the decision of where to segment the offered stream, is the main motivation for the TCP sequence space system. The sequence system is applied independently to each side of a connection as follows.

Consecutive bytes in the stream are assigned consecutive sequence numbers from a large ring of sequence numbers. The ring of sequence numbers is large enough (32 bits), so that the recycling of old numbers shouldn't pose a problem for a long-lived connection, because segments with a given sequence will have had a relatively long time to die out before their section of the sequence space is reused.

Hence, the reuse of sequence space only causes problems in reference to a new connection. When a new connection is created, an initial sequence number must be chosen. Use of a constant initial value is discouraged, because of potential problems if two processes open and close a connection in rapid succession. In this case, its possible that a segment from the first incarnation of the connection will live across the closing and reopening of the connection. If the same initial sequence number were to be used, this error is more likely.

To deal with this problem, TCP chooses initial sequence numbers based on a 32-bit timer. If the timer increments faster than data bytes can be placed on the medium, segments from previous connection incarnations will always have old sequence numbers, so long as the sequence space doesn't wrap around. If the timer period is longer than the segment lifetime, vulnerability is limited to times when the timer "laps" the sequence space, and then only if a connection is closed and reopened at that instant. The TCP specification uses a 32-bit sequence space and recommends that the (possibly fictitious) timer used to generate initial sequence numbers is incremented every 4 microseconds. This choice means that the system will work so long as the sustained data octets are generated less often than 4 microseconds (250 K octets/sec or 4 Mbps), and segment lifetime is less than approximately 2 hours (assuming half the sequence space is "new" and half is viewed as "old").

In addition to dat bytes, sequence numbers are assigned to two control items. In this way the control items, like data bytes, are protected from duplication or loss.

The two control items so protected are the SYN and FIN control signals. The SYN control signal is always the first sequenced item transmitted on a connection; the FIN is always the last.

The user supplies a push flag (PSH) with the other parameters of the Send call. The PSH flag controls the TCP server's options in buffering the data. If PSH is not set, the TCP server is free to buffer the offered data in the hope of combining it with data in subsequent send calls. Larger and less frequent transmissions will result, improving the efficiency of the connection. If PSH is set, the data specified by Send, along with any previously buffered data, must be formatted into segments and transmitted. When the TCP server at the other end of the connection sees the PSH flag in the segment, it must complete an outstanding receive without waiting for the receive buffer to fill. Logically, PSH is used to signal the end of a transaction and to insure that processing will begin. A later PSH may hide an earlier one.

Establishing a TCP Connection

Processes request a connection with a matched set of Open calls; the two TCPs involved establish the connection through a protocol known as the "three-way handshake."

There are two types of Open calls: an *active* Open that instructs TCP to try to establish the connection immediately, or a *passive* Open that instructs TCP that the process is willing to accept a connection if one is requested by another process. An active Open results in segments being sent and hence must specify the address of the other end of the connection; a passive Open can accept calls from any process, or may restrict the call to come from a specific process.

The basic purpose of the initial connect protocol is to guarantee that both ends of the connection want the connection and that both ends of the connection know the initial transmit sequence numbers so that reliable data transfer can take place.

TCP specifies the initial sequence value it will use for transmitting by sending a segment with the synchronize (SYN) bit set. The SYN is conceptually the first sequenced item sent over the connection, and it is never sent again, so the sequence field of the SYN segment corresponds to the sequence of the SYN and hence the first sequence value on that side of the connection. A TCP acknowledges that it wants the connection by acknowledging the opposite TCP's SYN and by transmitting a SYN segment of its own.

Each end of the connection believes that the connection has advanced to the established state, in which data transfer can begin, when the following three conditions have been met:

1. It has transmitted a SYN to its opposite.
2. It has received an acknowledgment for its SYN.
3. Its opposite has also supplied a SYN segment.

The initial connect protocol is known as the "three-way handshake" because the above conditions usually require a three-message sequence:

1. A segment from the actively opened TCP (TCP A) to the passively opened TCP (TCP P), carrying a SYN and initial sequence for the TCP A to TCP P side of the connection.
2. A segment from TCP P to TCP A carrying an acknowledgment for the first message's SYN, and a SYN and corresponding sequence number for the TCP P to TCP A side of the connection.
3. A segment from TCP A to TCP P acknowledging the SYN for the TCP P to TCP A side of the connection.

Although this mechanism is oriented toward a pair of calls, one active and one passive, it also works for a pair of active `Open` calls. One more message is necessary in this case because there are two SYN bearing segments and two acknowledging segments.

TCPs that wish to refuse a connection attempt do so using the reset mechanism described in the next section.

Abnormal Connection Termination

A TCP server can unilaterally close a connection by sending a segment bearing a reset (RST) flag. In general a RST is caused by an `Abort` call by the user or the arrival of a segment that couldn't possibly be correct. Some examples of conditions that result in a RST are the following:

1. Segments sent to nonexistent ports. This condition is often caused by hosts that crash and come back up. As soon as the other end of the connection sends a message to the now nonexistent connection, a RST is sent in reply, clearing the connection.
2. Segments which acknowledge data that was never sent.
3. Attempts to establish a connection with a passively opened port that isn't willing to accept the particular sender.

The decision whether or not a RST is warranted involves a tradeoff between the desire to catch all errors and the desire to preserve connections that can possibly be saved. For example, a TCP will usually not send a RST if its opposite sends data outside the window, as this could easily be the result of a badly delayed segment or an attempt by the opposite TCP to anticipate a window allocation. On the other hand, the RST criteria must be designed to prohibit an endless sequence of erroneous transmissions. Many of these choices are left to the individual TCP implementer. The only absolute restriction is that a RST should never be sent in response to another RST.

The receiver of a RST validates it by verifying that its sequence field corresponds to a reasonable value. In the case of a RST received during initial connection, the RST is valid if the RST segment acknowledges the appropriate SYN.

A valid RST is signalled to the user of an open connection, or a user attempting to actively open a connection, via a `Connection_reset` event. The TCP regards the connection as closed. If the connection was being passively opened, the RST returns the connection to the Listen state to await a new initial-connect attempt.

Normal Connection Termination

Connections exist for as long as they are useful to either end of the connection, or until the connection is unilaterally terminated due to error or some other abnormal condition. Thus the processes using a connection need a mechanism whereby they can signal that they have no further use for a connection. The rendezvous of two such signals implies that both ends of the connection are in agreement that the connection is useless, and hence that the TCP servers in question are free to "forget" about the connection and to release all resources allocated to the connection. TCP implements this service by having the two parties to a connection exchange sequenced control items, called FINs. The exchange is analogous to the exchange of SYNs that opens a connection.

A process signals its TCP that no more data will be transmitted on a connection (from its end) with a `Close` call to TCP. In response to the `Close` call, the TCP sends a FIN control down the connection. The FIN is sequence protected, so it will be acknowledged. Thus a TCP knows that both ends want to close a connection when the following have taken place; the fourth condition is implicit:

1. The TCP has sent a FIN.

2. The TCP has received a FIN from the other end of the connection.

3. The TCP has received an acknowledgment of its own FIN.

4. All data has been delivered.

The only ordering imposed on these events by TCP is that all data for a given side of the connection be sent before a FIN can be sent on that side and that a FIN must be sent before it can possibly be acknowledged.

The first constraint means that the last sequenced item sent over a given side of a connection must be a FIN. However, the two sides of the connection are independent. Thus one side of a connection may be closed, with a FIN sent and acknowledged, while the other side of the connection continues indefinitely.

The connection states associated with connection closing are those below the Established state in Figure 3-7. The complexity of these states is in part due to the use of state encoding to record the transmission of FINs and their acknowledgments and in part due to an attempt to improve the speed of connection closing in the presence of the "last message problem."

The "last message problem" refers to the fact that any message dialog between two parties must have a last message. By definition, this message isn't acknowledged, because the acknowledgment would then be the last message. Because this last message isn't acknowledged, the sender of the last message doesn't know if it reached its destination, and a process expecting such a message doesn't know if the message was sent unless the message arrives. Because of these constraints, the last message can only be protected by means other than a positive acknowledgment, i.e., a passive acknowledgment such as is used in timer-based systems [Watson 77, Watson 79] or through the use of probabilistic arguments and multiple retransmissions.

In TCP, this problem manifests itself in regard to the acknowledgment of FINs. The TCP state machine in Figure 3-7 attempts to solve the problem by specifying two different control algorithms for the TCP that sends the last FIN and the TCP that acknowledges the last FIN. The sender of the last FIN arrives at the Closed state through the Closing state, while the acknowledger of the last FIN passes through the chain leading through Time Wait. The intent is that the sender of the last FIN knows when the connection is complete (because it gets an ack for the last FIN), whereas the acknowledging TCP can't be absolutely sure that its ack got through. Therefore the acknowledging TCP waits for a period equal to twice the maximum segment lifetime (MSL) before it deletes the connection (knowing that the sender will persist in retransmitting for a period equal to MSL). This logic works for cases in which segments aren't lost, or if the two ends close the connection at significantly different times.

Suppose that the two ends of the connection decide to close the connection at approximately the same time. Both send FINs, and when the FINs arrive, both send acks for the other's FIN. At this point in the closing protocol, both TCPs have advanced to the Closing state, and two acknowledgments are in the network, travelling in opposite directions. If either of these acknowledgments is lost, the result is a connection that has been deleted at one of its ends, and which has a TCP awaiting an acknowledgment at the other end of the connection.

The end that has closed has closed normally, in the knowledge that all was well. At the other end of the connection, the FIN will eventually be retransmitted. This retransmitted FIN will arrive at a TCP that no longer recognizes the existence of the connection, so it will result in a RST segment being sent in reply. The RST will finally close the other end of the connection, but in an unsatisfactory manner; the two processes that were communicating have different opinions whether the connection terminated normally. Note that if the FIN was piggybacked on a data segment, there is also no way of telling whether the data was delivered or not.

When the RST arrives, the TCP doesn't know what has happened. One possible interpretation is that the segment arrived at a host that responded with a RST because of a restart. Another possible interpretation is that the RST was sent in response to the arrival of a duplicate segment after the segment had normally closed at the foreign end.

Two modifications of the protocol are thus suggested for any implementation:

1. FINs should probably not be piggybacked on data segments.
2. Both ends of a closed connection should probably wait for some timeout approximating a MSL before deleting connection data after the connection has satisfied the criteria for closing.

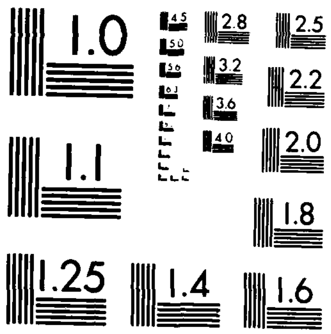
3.5 TCP IMPLEMENTATION OUTLINE

Design Considerations

The main computational cost for TCP on a timeshared host is operating system service for functions such as context switching, scheduling, and resource management. In the study of a TCP server under UNIX [Bunch 80], 72 percent of CPU time was consumed by OS kernel code, exclusive of TCP server per se. Thus when TCP servers are implemented for these hosts, they are not structured like the clean multiple process models used for conceptual discussions. Instead, TCP

AD-A125 441 COMMUNICATION ENVIRONMENTS FOR LOCAL NETWORKS(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST P V MOCKAPETRIS DEC 82 ISI/RR-82-103 MDA903-81-C-0335 2/3

UNCLASSIFIED F/G 17/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

code is combined with code for IP and other layers, and the overall implementation is structured to fit the I/O service model for the host's operating system.

When the TCP server is moved into a front-end processor, the server's efficiency can be improved simply by supporting the server with a spartan operating system which is tailored to the protocol task. In Bunch's study, such a change led to a kernel which was 7 times as fast, yielding a 3-fold improvement in throughput. Kernel time was reduced to 32 percent of the total CPU cost. A less tangible advantage is that the server can be designed to fit the protocol definition rather than the host operating system.

The key design issue for front-end systems is the path between the host and the network interface. This path should ideally be general enough so that dissimilar hosts can use the same network interface, fast enough not to form a bottleneck, and reliable enough so that network communications are not made significantly less reliable.

One approach to the problem is to use a loosely coupled path and a new host-to-front-end protocol. This is the approach used in [Bunch 80]. The new protocol duplicates some TCP functions such as flow control, sequencing, and retransmission. While this approach adds some reliability and architectural freedom, it also recreates part of the host load that was used to justify the front end in the first place.

The alternative is a tightly coupled interface which transfers processed data directly with the host and relies on bus parity or similar facilities for reliability. While this design is almost universal for low-level interfaces, using it for high-level interfaces is criticized on the grounds that the path between the host and front end is insufficiently reliable and hence destroys the end-to-end nature of error control. Superior reliability is claimed for schemes which pass packets all the way from their source to their final destination without recomputing error control fields. (Incremental alterations for hop counts, etc., are excepted.)

This objection has been cited regarding the 1822 interface for the ARPANET IMPs in that the high-speed serial link between the IMP and the host is not protected as throughly as the IMP-to-IMP links. The new Xerox Ethernet standard explicitly recommends that packet checksums be verified by hosts and not by interfaces, although it also mentions that additional link-level error control is acceptable as long as it stays confined to the link layer.

The end-to-end argument is persuasive for gateways and other forwarding nodes, but is weaker when applied to a network interface at the final destination. Here the increased possibility of error due to additional components in the critical path must be balanced against the lower chance of failure due to less complicated and better protected environment in the front end. The net loss in reliability must be compared against the gains in architectural freedom; because protocol actions must be delayed until error control is complete, little distribution of function is possible if the end-to-end approach is followed. The end-to-end approach also has a lower performance limit than a distributed approach.

Proposed Server Structure

The proposed server structure violates the end-to-end principle and relies on a tightly coupled interface between the host and the network interface. The structure justifies a theoretically lower reliability by offering higher performance, lower host load, and a distributed structure which might eventually benefit from advances in LSI.

The server is organized in sections which correspond to the three external events that trigger server actions:

1. *User interface* logic which handles requests for TCP services from processes. Most of this logic resides in the host; a small amount in the network interface is intended to be the only host-dependent code in the network interface.
2. *Segment arrival* logic residing in the network interface which is activated by packets arriving on the network and timeouts when expected packets fail to arrive.
3. *Segment transmission management* logic in the network interface which formats packets and schedules their transmission and retransmission.

This discussion omits the underlying protocol layers; they are assumed to add logic similar to that discussed for the DCS message-level implementation.

These sections communicate through a shared data structure which contains the state, pending user requests, and buffers for all connections. Although the data could be located in the host, placing it in the network interface is a better choice because of the more frequent access by the interface and because it leads to simpler hardware in the interface. The state data and buffers are derived from the TCP standard and hence can be made insensitive to host features; the request queues are clearly host dependent.

Connection State

A TCB containing the state data required by the TCP standard is shown in Figure 3-11. Several values are represented differently from the conceptual model of the TCP standard. In particular, SYNs and FINs are explicitly recorded rather than being encoded in the connection state.

Connection-Request Queues

When a user issues a TCP request which can't be completed immediately, the request is recorded in a request element attached to the corresponding TCB. The request element contains the following:

1. The arguments of the call as supplied by the user.
2. Host dependent structures which describe an action corresponding to every possible request outcome. For example, a `Send` request could have two possible outcomes: success and failure due to any cause. A `Send` request would then specify two different actions corresponding to these conditions. For example, a `Send` request element might specify a pseudointerrupt if the request fails or is timed out and no operation if it completes normally. The *number* of actions is host independent; the exact fields required are host dependent. The typical contents of the host dependent values would include a process handle and an interrupt channel.

```
TCB=record

{ Common data }

 local_connection_ID:half_word {handle for connection}
 local_socket:socket           {socket for this end of connection}
 foreign_socket:socket,        {socket for other end }
 backup_foreign_socket:socket, {foreign socket for listen restart}

 state:connection_state,       {one of Opening, Established, Closing}

 active_open:boolean           {was connection OPENed actively ? }
 FIN_in,FIN_out,FIN_acked,     {switches for state transition control}
 SYN_in,SYN_out,SYN_acked:boolean

{ Send side data }

 snd_left,                     {Oldest sequence not acked}
 snd_user,                     {sequence of next octet from user}
 snd_limit,                    {sequence of first octet beyond window}
 snd_urgent:sequence;          {sequence of urgent octet}
 snd_max_size:integer,         {maximum segment size to transmit}
 snd_urgent_present:boolean,   {is there a pending urgent}
 snd_queue:pointer;            {request queue for transmission}

{ Receive side data }

 rcv_SYN_sequence,             {sequence of received SYN}
 rcv_user,                     {sequence of oldest buffered octet}
 rcv_left,                     {sequence of first unreceived octet}
 rcv_limit,                    {sequence of first octet beyond window}
 rcv_urgent:sequence;          {sequence of urgent octet}
 rcv_urgent_present:boolean,   {is there a pending urgent}
 rcv_queue:pointer;            {request queue for reception}
```

Figure 3-11: TCP Transmission Control Block (TCB)

3. Scheduling data for use within the network interface to control timeouts, etc.

Connection Buffers

The traditional purpose of buffering is to mask transient mismatches between rates of production and consumption. In communications, buffering procedures must be designed to accept great variations in rates and to cope with arriving packets which contain data which is out of order, overlaps previous data, and may be erroneous. In this proposal, data buffers are used as an intermediate data structure which also helps to decouple segment-processing routines from the host-oriented user interface.

The proposed scheme dedicates two circular buffers to each connection. The buffer size is fixed at initial connect time. The data in the buffer is designated by a start pointer, a length, and a pointer which identifies the newest data octet to be associated with a PSH. Arriving data is inserted into the buffer if it is new, contiguous to the previously buffered data, and does not "wrap" the circular buffer. If no data octet in the buffer is associated with a PSH, the PSH pointer takes a special value. Only the most recent PSH can be remembered; this follows the spirit of the TCP definition.

This scheme is by no means the only possible alternative; several others were discarded. Linear buffers similar to those seen in the TOPS-20 TCP implementation were discarded because they complicate window management. To see this consider the choices with a linear buffer that is not filled but terminated with a PSH. If the window allocation was based on the length of the available buffers, then the window will shrink by the size of the buffer and not by the amount of data received; shrinking windows are explicitly discouraged in the TCP standard because of the induced flow control oscillations. Data consumption from the buffer poses a similar problem unless recompaction is used, in which case the linear buffer is simply simulating a circular one.

Buffers in the host's memory have several potential advantages. Resource allocation can be the responsibility of the host or even of host processes; this allows the amount of buffering to be better matched to the process. If data is placed directly into host buffers as it arrives from the network, an extra copy operation is eliminated. However, the segment-processing routines must now be configured to match the host's addressing conventions; different address sizes, real-to-virtual address mapping, and page boundaries are all potential problems. The cost of the additional copy is probably insignificant. The cost of all primitive data operations on the segment was measured to be less than 10 percent [Bunch 80]; the same experiments were unable to detect a measurable difference in CPU time for segments with different amounts of data. For these reasons an intermediate buffer in the network interface is selected; a possible compromise is to map interface buffer memory into the address space of the host when interface design and host hardware make this possible.

Buffering data which is newer than already received data but not contiguous to it is desirable but difficult to do in practice. A bit map for octets in the buffer, or a set of ranges would be required and would complicate processing without offering any benefits in the local network environment where there is usually no advantage in allowing multiple transmissions to be in progress. For connections to distant networks that might benefit, out of order segments could be queued from the TCB for later reprocessing.

User Interface

The user interface handles the requests for TCP services made by processes. These requests are functionally equivalent to those listed in Table 3.2, but may be restated to conform to the OS conventions of the host.

The first step in processing any user call is to associate it with a connection and its TCB. This step includes TCB creation for new connections and also includes validation of access to port IDs and connections according to host policies. The user process identifies the connection either by specifying both sockets or by handle.

Once the TCB is located, control passes to a routine for the specific request type (e.g., `Receive`, `Send`, `Status`). The routine performs that portion of the request which can be completed using TCB data. In some cases, the user call can be immediately completed. For example, a `Status` request can always be completed using TCB data; a Receive request will complete immediately if sufficient data is in the receive buffer.

In other cases, the request needs data which is not in the TCB. For example, an active Open can never complete until a segment is sent to the other end of the connection and a reply returned; passive Open calls often wait for hours or days for a caller; Receive requests may have to wait for an indefinite period for data from the other end of the connection. In these cases, the processing routine constructs a request element and chains it to the TCB.

The processing routine is reactivated whenever an arriving segment changes the TCB state in a way that might influence the pending request or when the timeout of the request expires. The request will eventually complete due to timeout or segment arrival; transmissions never cause reactivation because they either simply reflect the existence of buffered data, in which case the request would have completed when the data was copied, or the actual transmission must be confirmed with a returning acknowledgment.

Segment Transmission

Transmission scheduling is a resource allocation problem. Every node in the network allocates bandwidth, in the form of segment transmissions, to its own connections. Because nodes act without complete information, the segment transmission code is always a combination of heuristics intended to balance throughput, reliability, delay, and other metrics, both for individual connections and the network as a whole. Servers typically have two sorts of heuristics: those for resolving conflicting requests from separate connections, and those which limit the rate at which a connection makes requests.

Arbitration between multiple connections is generally designed to achieve equal access. Equal access at the host level is guaranteed by the medium's access-control system. Within a host, equal access is usually provided by a first-come, first-served algorithm covering connections which need to transmit, or by a round-robin polling of all connections.

The reliability, throughput, and delay characteristics of an individual connection are almost always enhanced if the connection can transmit segments more often; however, the degree of benefit varies greatly for different connection states and histories. Individual connection management heuristics limit transmission requests to those which have the greatest chance of being useful, i.e., of delivering new, rather than already received data and control values.

The major heuristics for individual connections are those which select retransmission timeouts. Retransmissions are required because segments, or the acknowledgment segments they cause, can be lost. The timeout should approximate the amount of time that must elapse before the sender can be fairly certain that the acknowledgment will never arrive. This estimate must include the transit time of the segment, the transit time of the acknowledgment, and the time required to process the segment at the other end of the connection. Different connection states require different timeouts; if a connection is transmitting into a zero window in hopes of a window update, its retransmission interval should include time to run the consumer process.

Other heuristics attempt to batch control value changes and new data, rather than requesting a transmission each time a control value changes or new data is made available. This idea is explicitly represented in the PSH mechanism, but can also be extended to control by inserting a timeout between the time a control value changes and the time that a transmit request is made. Using this scheme, when a process receives a segment containing a query, and hence changes the connection's acknowledgment sequence, the process has time to calculate a reply which the server can combine with the acknowledgment. This scheme trades increased delay for lower bandwidth.

An important batching heuristic is to delay formatting of actual segments until the last possible moment before they are actually transmitted on the medium. This guarantees that each segment carries the most up-to-date control values and data octets. The drawbacks of this scheme are that a segment (or at least its header) must be recreated every time it is to be transmitted and that the TCP server can't create a large queue of segments for IP or another lower level to transmit, but should instead format a segment each time a transmission completes.

By varying the selection of heuristics and the complexity of the algorithms which estimate delays and arrival rates, the designer can create transmission control systems with almost any level of complexity. The best system for a given server depends on the server's need for more complicated control (and presumably greater efficiency), versus the cost of the system in terms of CPU time, delay, etc. Simpler facilities are justified in cases when usage is easily predictable or completely unpredictable, when transmission times and service times are insignificant or regular, or when the server simply can't afford the calculations. More complex systems are beneficial when delays are highly dependent on the identity of the destination host, when bandwidth is expensive compared to computation, or when efficiency must be maximized.

Using the buffering discipline already discussed, a suitable connection retransmission schedule can be implemented using a transmission scheduler which continuously scans all TCBs, and a single decrementing timer per TCB. A zero timer value means the connection is requesting a transmission, and will get it the next time the scanner services this connection. (In an actual implementation, the timers will probably contain values from a ring of numbers and the scanner will avoid the continuous overhead of scanning by maintaining some sort of data structure which keeps the TCBs ordered by time.)

Timer management is embodied in the set of rules for setting the timer value. Different timeouts are associated with different transmission states, where the state can be deduced from TCB values and flags which are set when control values are updated. The proposed scheme uses four different timeout values. In order of decreasing length, the values are as follows:

1. A long-term timeout which is set when the connection is idle, i.e., there is no data to transmit, no control values have changed, and no data-bearing segments have arrived from the other end of the connection. The purpose of this timeout is to detect inactive connections to restarted hosts.
2. A timeout value for cases in which there are no control updates or segments from the other end to acknowledge, but there is data to transmit and a zero window for transmission. This timeout is necessary because window updates are not themselves protected by retransmissions, and hence the server has no way of knowing for sure whether the other end of the connection has a closed window. TCP servers deal with this problem by periodically sending a segment, *even without an allocation*, so that lost window allocations don't hang a path.
3. A timeout corresponding to the case in which the server is waiting for an acknowledgment of data transmitted with a window allocation, but no control value changes or data-bearing segments have been received since the last transmission.
4. A timeout which is set whenever a control value is changed, new data is available for transmission into an open window, or a data bearing segment has arrived from the other end of the connection. This timeout is not intended to delay the action per se, but rather to allow multiple events to be handled by a single transmission.

Although these timeouts could be constants within the local network environment, suitable generality for an internetwork environment could be created by associating each TCB with one of several timeout sets. This feature adds another TCB field.

Segment Arrival

The code which processes arriving segments is unique among the parts of the TCP server in that it must process segments constructed according to any interpretation of the TCP standard, as opposed to the other sections which can implement a single legal policy. For example, the transmission side is free to never combine SYNs, data octets, and FINs in a single segment; the code that processes arriving segments can not make a similar assumption. This generality is part of the "robustness principle" advocated in the TCP standard: "... be conservative in what you do, be liberal in what you expect from others" [Postel 81a].

The effect of this policy is that a small fraction of the segment-arrival code processes "normal" cases, and a large fraction of the code is used infrequently for processing special cases and unexpected conditions.

The segment-arrival code has several aspects which are not completely defined by the TCP standard. For example, various servers are tolerant to a different degree in deciding what is a fatal error and what is forgivable. In one server, data octets outside of the window might provoke a RST whereas they might be ignored in another. Different servers have different policies for deciding when a sequence value is old, new, or possibly in error.

The behavior of the segment arrival code is well defined for reasonable cases. Figure 3-12 illustrates a possible implementation of the top-level logic. The `Segment_arrives` procedure is internally divided into three sections, represented by the three top-level IF statements. `CheckWindow` tests to see if its arguments are ordered by a greater or equal relationship in the sequence space. The details of computing `CheckWindow` are discussed in [Plummer 78].

The first step in `Segment_arrives` is to call special code to process the segment if the connection has not yet reached the established state. This code is shown in Figure 3-13 as procedure `Conn__init`. This code treats several segment fields differently from the similar code in `Segment_arrives`; in particular, the acknowledgment-sequence value must match that of the transmitted SYN, and RSTs are treated differently.

The `Conn_init` procedure can leave the connection in the Opening state, reset the connection, or advance the connection to Established state. Before advancing the connection to Established state, `Conn_init` sets up both sequence spaces.

The second IF statement in `Segment_arrives` processes RST bearing segments that arrive at an established connection. The segment is ignored if its sequence number suggests that the segment is from a previous incarnation of the connection. This test is represented by the function `reasonable_sequence` (not described herein). There is no standard way to perform this test; detecting unreasonable sequence numbers requires assumptions about the bandwidth and maximum packet lifetime of the network. Some TCP implementations assume that sequence ordering can only be new or old, and never unreasonable. Note that sending a RST in reply to a RST is never allowed; this rule prevents an infinite sequence of RSTs.

```
procedure Segment_arrives;

   if state=Opening {if not yet established do initial processing}
   then Conn_init;


   if state>Opening & (RST in seg.flags)
   then process_reset;


   if state>Opening {if open process segment, otherwise ignore}
   then if (SYN in seg.flags) & not(seg.seqno=rcv_SYN_sequence)
        then generate_reset
        else begin {process a normal segment}

             if OPT in seg.flags
             then process_options;
             if (ACK in seg.flags) &
                CheckWindow(snd_left,seg.ackno,snd_user)
             then process_ack;
             if (URG in seg.flags)
             then process_urgent;

             if New_data()
             then begin
                  process_data;
                  if FIN in seg.flags
                  then process_fin;
                  end;

             if (SYN in seg.flags) ! (FIN in seg.flags) !
                seg.data_length<>0
             then schedule_ack_trans

             end

end; {Segment_arrives}
```

Figure 3-12: TCP segment arrival code

The last part of Segment_arrives processes "normal" segments after filtering out segments from previous connections. The tasks for normal statements are represented by the statements in the begin block.

Options

The present TCP standard defines a single option other than padding and end of option list codes. This option is valid only in initial-connect segments, i.e., when accompanied by a SYN. Hence, no option processing is as yet required.

```
procedure Conn_init;

   if RST in seg.flags {if reset present validate it }
   then if SYN_out & (ACK in seg.flags) & (seg.ackno=snd_user)
        then process_reset
   else begin

        {test for ack for our SYN}
        if ACK in seg.flags
        then if SYN_out
             then if seg.ackno=snd_user {ack matches our SYN}
                  then begin
                       snd_left:=seg.ackno
                       SYN_acked:=true
                       end
                  else generate_reset {fraudulent ackno}

        {test for arriving SYN}
        if (SYN in seg.flags) & (not SYN_in)
        then begin
             setup_with_SYN
             if not SYN_out then send_syn
             SYN_in:=true
             end

        {if all conditions met set state to established}
        if SYN_in & SYN_acked
        then state=Established

        end

end; {Conn_init}
```

Figure 3-13: TCP initial connect code

Acknowledgments

Procedure `process_ack` is called if ACK is set and the acknowledgment sequence value in the segment covers previously unacknowledged data. This procedure should update the TCB value and complete the corresponding `Send` requests.

Window

If ACK is set, the window sequence value is computed and compared to that in the TCB in `process_window`. A new window value may allow previously queued data to be transmitted.

Urgent

If URG is set, the urgent sequence of the segment is computed. This value generates an urgent condition if there was no previous urgent condition pending and replaces a previously pending urgent if its sequence is newer.

Data octets and PSHs

If any sequenced items in the segment (including both data octets and FIN) are in the receiver's window they can be copied, although many servers refuse sequenced items which are not contiguous to previously received items. The test for this condition is complex since the receiver must be able to extract desirable data octets (if any) from segments which may precede, follow, or in any manner overlap the receive window, and must adjust all data octet sequences depending on whether or not the SYN flag is set. The test is represented here by `New_data`.

FIN processing

If an arriving segment carries a FIN, and all sequenced items up to the FIN have been received and buffered, then the `process_FIN` is called to begin the process of closing the connection. This code is very similar to `Conn_init`.

Acknowledgment transmissions

If any sequenced items were contained in the segment, the TCP server queues an acknowledgment via `schedule_ack_trans`. The server may delay this acknowledgment so that it can be combined with new data or control value changes, or even ignore some portion of such requests, but the transmitter must be guaranteed an ack if it retransmits enough.

Distributions

The servers studied in [Bunch 80], the pseudocode descriptions, and TCP servers for Multics and TOPS-20 all have several consistent properties:

- Operating system costs dominate unless extremely large buffers are used. In most existing networks, TCP segments are restricted to 512 data octets by convention to avoid IP fragmentation, and hence operating system costs do dominate.
- Within the TCP code *per se*, most processing is concerned with deciding what to do, rather than in costs for data movement, checksumming, etc. Most of these decisions are sequence decisions.
- Optimal, or in some cases even acceptable, performance requires a tuning of timeouts, buffer policies, and acknowledgment strategies. The best policy is not always that which transmits control value updates the most often; for example, when a transmitter generates data faster that the receiver can accept it, a frequent acknowledgment and retransmission policy results in many small data transmissions and acknowledgments. Similar conditions can be triggered by buffering policies, inappropriate PSH segments, and other conditions. In practice, the so-called "silly window syndrome" can produce throughput an order of magnitude lower than that achieved by a less eager algorithm and always increases host CPU time requirements by a similar amount.

4. A MODEL ARCHITECTURE

4.1 INTRODUCTION

The previous chapters of this report surveyed the state of the art in communication protocols and local network technology and mentioned problems and opportunities for specific examples. This chapter defines areas for research in interface design and develops an interface model. The areas define an agenda for the rest of the report; the model is a framework for evaluation in terms of cost, effectiveness, and practicality.

The areas are intended to select aspects of communication processing where protocol implementation tools tailored to communications would be of value. The areas are functional in nature, so that they will be as independent as possible from specific interface technologies, communication protocols, and communication loads. This mandates a "top down" approach to area definition that addresses problems present in a wide variety of communication environments. The areas are defined in the "goals" section of this chapter.

The model is a top-level design of a new interface, together with the set of assumptions and alternatives regarding the model; the primary purpose of the model is to associate costs with the alternatives. The model interface can be implemented in a variety of technologies. The model's development distinguishes between issues related to the architecture, implementation, and realization of the interface, where these terms are used in a manner similar to that proposed for CPU design in [Blauuw 70]:

- *Architecture* refers to the functionality and interface organization that is visible to the host.
- *Implementation* refers to the internal structure, i.e., the actual interface components and their interactions.
- *Realization* refers to the actual technology used to build the interface.

4.2 GOALS

Communications systems are designed to meet varied constraints. In the case of local networks, the usual goal is to minimize interface hardware while preserving as much as possible of the inherently high bandwidth of local media. Software in the host, and possibly the interface, creates the perceived character of the communication environment and overcomes any shortcomings of the interface. The protocol software is the limiting factor which governs the end-to-end performance of the communication system.

The approach of this report is to acknowledge the superiority of the software approach in terms of cost, but to claim that higher performance and new types of service can be achieved by specialized protocol processing facilities in the interface. The crucial issues in designing these facilities are insuring adequate protocol flexibility and acceptable cost.

Protocol Flexibility

The typical local network host of the future will be connected directly to the local network and indirectly to one or more internetwork systems. The host will need to use a combination of protocols: one or more "standard" protocols for compatibility with internet systems (e.g., X.25, IP, TCP), and local protocols oriented toward simplicity and performance. The protocol flexibility needed by such a host has many aspects, including protocol changes and maintenance, addition of new protocols, and supporting the simultaneous use of multiple protocols. Even if this view is incorrect, and it becomes possible to use a single standard protocol at some level of the protocol hierarchy, flexibility in assigning policies for flow control, buffering, block size, etc., will be needed.

The key to designing protocol tools that give this flexibility is to create a set of simple protocol tools that address individual protocol algorithms and a control structure that allows tool composition. A protocol implementation is a "program" of directives to the individual tools. The top level of the control structure branches to allow several parallel protocols.

This approach relies on the similarities between protocols, but allows for differences. For example, all protocols detect duplicates using sequence spaces generated with modulus arithmetic, although the unit of sequence and modulus varies greatly. Similarly, the relationship between protocol algorithms varies. For example, most protocols have acknowledgments, duplicate control, and unilateral resets, but their order of evaluation and sharing of packet data varies.

Cost

The new protocol tools require additional hardware and, hence, cost. This cost penalty mandates careful selection of new tools to reduce the cost of individual interfaces and the local network as a whole. The following heuristics are proposed:

- Tools that provide higher performance implementations of existing protocols (e.g., TCP, IP) should recognize that the entire protocols are too complex to be implemented in a reasonable amount of hardware. A solution is to include a microprocessor in the interface to handle special cases (e.g., connection initiation and termination, urgent), errors, diagnostics, etc., and restrict the hardware tools to those necessary to implement simple primitives. The proportion of the protocol implemented in the microprocessor should be large in a static sense and small in a dynamic sense. Each transaction in the "automatic" subset should require the absolute minimum of host and microprocessor intervention, so as to minimize latency and maximize throughput. Microprocessor intervention should not be required for time-critical functions and simple data transfers.
- New services, and hence their tools, should take advantage of the local network medium and create a better environment for distributed computation. The envisioned environment sits between the current local network and multiprocessor systems and has the performance and simple interactions of the multiprocessor while preserving the loosely coupled organization of the local network. What is needed are more powerful methods for specifying interprocess communication connectivity and simple primitives for object transfer. These requirements can be met by better naming systems, hardware to simplify data transfer, and more sophisticated prompt acknowledgment facilities. Broadcast delivery with reliable prompt acknowledgments would be particularly desirable for implementing reliable distributed systems that need to keep synchronized databases.
- Any design will be more effective if it can transfer technology, in the form of hardware or

software, from existing systems. The interface should be designed to allow the use of existing LSI parts for the microprocessor, host DMA interfaces, etc.

- The overall cost of the local network can be minimized if it allows a variety of interfaces in the network. For example, large hosts on the network might require more facilities than terminals. Experience to date suggests that the cheapest interface is microprocessor based. Hence the difference between interfaces will be one of speed rather than intelligence. Schemes for prompt acknowledgments, addressing modes, etc., should anticipate the use of less capable interfaces. *In particular,* interfaces are not required to generate or receive prompt acknowledgments, although interfaces that do not generate prompt acknowledgments may be required to signify this lack in a special prompt ack value.

Area Definition

This report divides protocol processing, and hence the protocol tools, into two major areas: *binding* and *data delivery.*

In terms of a state model of protocol execution, binding refers to state representation, transition selection, state update, and multiplexing of multiple protocol state machines. Data delivery refers to the actions that are associated with transitions.

In an actual interface, binding tools relate stimuli such as arriving packets, user requests, and timeouts to a description of the connection state and generate control signals for state update and the data-delivery tools. For example, in processing a DCS style packet, the binding tools might recognize a well-formed packet (good CRC, all fields present, etc.), look up the relevant connection record or records, filter the packet by sequence, and output connection-record information to control the transfer of packet data into host memory and possible host interruption.

Data delivery tools are driven by control information output by binding tools. The problems in this area are the definition of the data objects carried by the medium and methods for transferring these objects between the interface and the host. Most host-specific considerations, such as byte sizes, memory addresses, buffering policies, and reassembly, are isolated in this area.

4.3 ARCHITECTURE

The proposed interface model is shown in Figure 4-1. The figure is the host's model of the interface and does not necessarily represent the actual interface structure.

Binding functions reside in the filter and the microprocessor. The status and control registers are a register file used to pass commands and status between the microprocessor and the host.

- The filter has two major components: storage for the binding specifications and connection-record data, and an interpreter that uses the specifications to perform the binding functions. The binding functions performed by the filter include name recognition, connection-record search and storage, state update, and prompt acknowledgment control.
- The microprocessor is a conventional computer, whose program is dedicated to

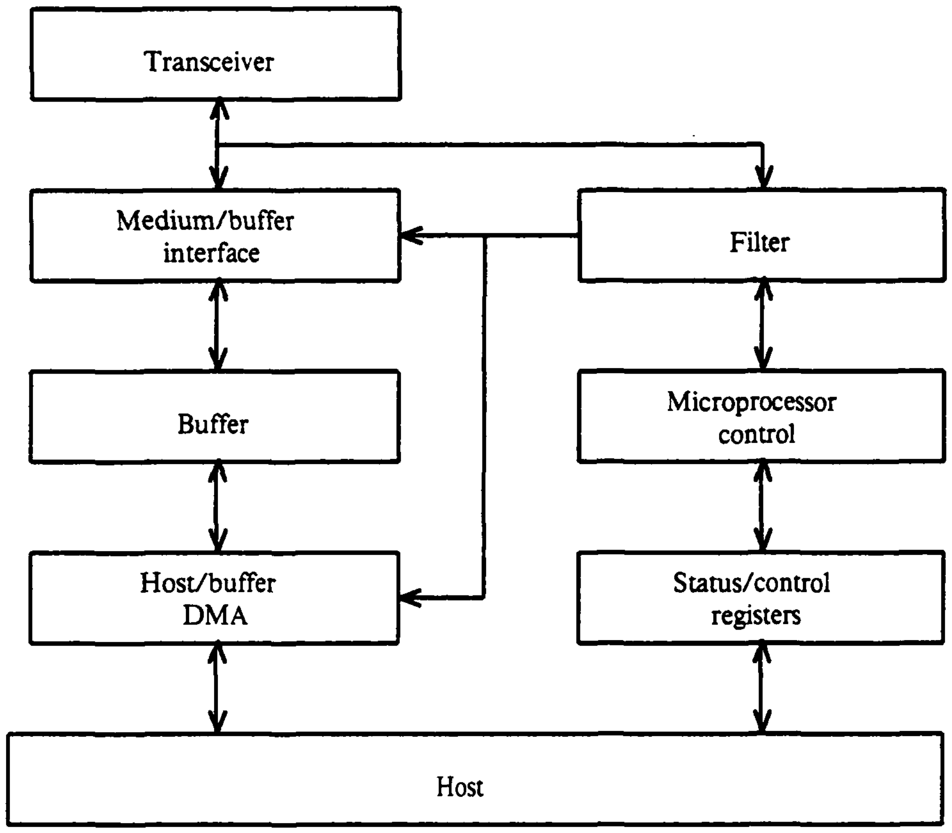


Figure 4-1: Architecture for the model interface

maintaining the database stored in the filter and executing the parts of protocols that are not performed directly as the result of filter operations.

The boxes on the left side of the figure represent the main data paths:

- The transceiver includes the transmission line interface, isolation, and logic for transmitting prompt acknowledgments generated by the filter. If possible, the transceiver should be the only part of the interface that depends on a particular medium.
- The medium-to-buffer interface converts between the serial format used on the medium and the parallel format used by the buffer. Its operation is controlled by signals originating in the filter.
- The buffer is a multipacket buffer used for retransmission and incoming packets.
- The host-to-buffer interface transmits packets between the host memory and the interface buffer and probably includes logic for generating host interrupts. Its operation is controlled by the filter.

There are three types of interface activity: host commands that direct interface operations, message reception, and message transmission. The following sections roughly outline the operation of each of these activities.

Host Command Processing

The host presents commands and associated parameters to the interface via the shared memory in the status and control registers. The microprocessor looks for complete commands in the registers as part of its idle loop.

Commands are instructions for updating the filter database. The microprocessor synchronizes the actual filter database update with respect to message arrival and transmission, so that a partially updated database is never used to make reception or transmission decisions. Once the update is complete, the host is informed by a status update and an optional interrupt.

The same data path may be used for downloading of the microprocessor program.

Message Reception

The start of a new packet is detected by the transceiver, which recognizes the preamble or epoch at the start of the transmission. The transceiver signals the filter and data path elements to start processing the incoming data. The arriving data are bit unstuffed, if required, and passed to the filter and the medium-to-buffer interface. The packet is stored in the buffer as it arrives.

The filter first decides if the incoming packet is destined for the attached host. If not, the transfer to the buffer is aborted, and the transceiver is signalled to begin searching for a new packet. If the packet is recognized, the filter searches its database for the corresponding connection record.

If the connection record can't be located, a microprocessor interrupt is generated, and no automatic processing takes place.

If the connection record can be located, the filter processes control information in the header against data in the connection record. The exact processing to be performed is specified as part of the filter data structure. The evaluation results in control signals which control the data-delivery tools, may update the connection-record data, and may cause a microprocessor interrupt.

The exact combination of actions depends on the packet contents, and probably should not cause permanent change until the end of the packet arrives and the packet CRC is checked.

Once the packet is complete, the interface generates a prompt acknowledgment based on the filter control signals. The prompt ack is composed of fields which correspond to separate filtering decisions and ultimately to events of interest to the interface and protocol transmitting the packet. For example, one field might signify that the CID was recognized, another the fact that the packet was received with good CRCs and checksums. The meaning or validity of ack fields may depend on other ack fields; in the example, the CID recognition is probably meaningful only if the CRC and checksums were correct. The meaning of ack fields will also be dependent on the layering of packet data. For example, TCP ack fields would probably be in different positions if both IP and some local protocol were used to encapsulate TCP segments.

Message Transmission

Requests for transmission ultimately originate in the host, although retransmissions and packeting of long messages can originate in the microprocessor. Packet transmission requests are queued in connection records in the filter database.

As part of its idle loop, the microprocessor searches the filter database for transmissions that should now be scheduled. The search looks for the highest priority transmission which is not waiting for a timeout to expire. (Timeout management is remarkably similar to sequence-based searching as it is also based on modulo arithmetic.) Once an eligible transmission is located, the appropriate connection-record data is transferred to the host-to-buffer DMA, and the packet is transferred from the host to the buffer.

The packet is transmitted as soon as the medium can be acquired.

The result of transmission is returned status that includes transmission-error flags and any prompt acknowledgments that were returned. The status is used to update the connection record and may possibly cause a host interrupt or update of host memory.

4.4 IMPLEMENTATION

Design of the implementation level structure for the model is driven by the worst-case processing requirements of the architectural level. The processing requirements are of two general types: data transfer between the buffer and either the host or the medium, and control functions such as filtering, prompt ack processing, and data path setup. The worst case processing requirements, taken with the processing capabilities of various computing elements, define the maximum sustainable data rate for the interface.

The candidates for worst case are host interaction, packet transmission, and packet reception.

- Host interactions which require host memory speed response (e.g., interface device status) should be directly performed by the status and control registers; other host interactions (e.g., update of filter database) are not timing sensitive and can be performed whenever the interface is otherwise idle (i.e., when no packet is arriving or departing). Hence, host interactions do not cause worst-case behavior.
- Because packets to be transmitted can be preformatted in the interface buffer, and because the returned prompt acknowledgment need only be processed before the next transmission, the worst-case demands associated with transmission occurs as the packet is being output. This data-transfer demand is directly related to the data rate of the medium.
- Message reception has the same requirement for data transfer between the buffer and medium; in addition, filtering must take place. Thus the worst-case activity should be observed during message reception. (If the interface can transmit to itself, the worst case occurs during the overlap of transmission and reception. The peak load is the reception peak load plus the constant load generated by transmission host-to-medium transfer.)

The exact timing of data transfer and control requirements during packet reception will vary depending on the exact filtering discipline, prompt acknowledgment system, and data-transfer schemes used. Figure 4-2 gives the receive timing relationships based on the assumption that the interface constructs prompt acknowledgments based on packet contents.

The timings are divided into three types: *fixed* timings that are invariant, *optimal* timings based on an interface design with unlimited performance, and *maximum filter* times which are the lowest performance timings consistent with the assumptions. The horizontal axis is given in terms of the fields of a packet; the sizes of the fields vary widely. The sizes are given in terms of bits of packet data; the ranges correspond to the differences between a RI packet and a TCP segment encapsulated in an IP header (without options).

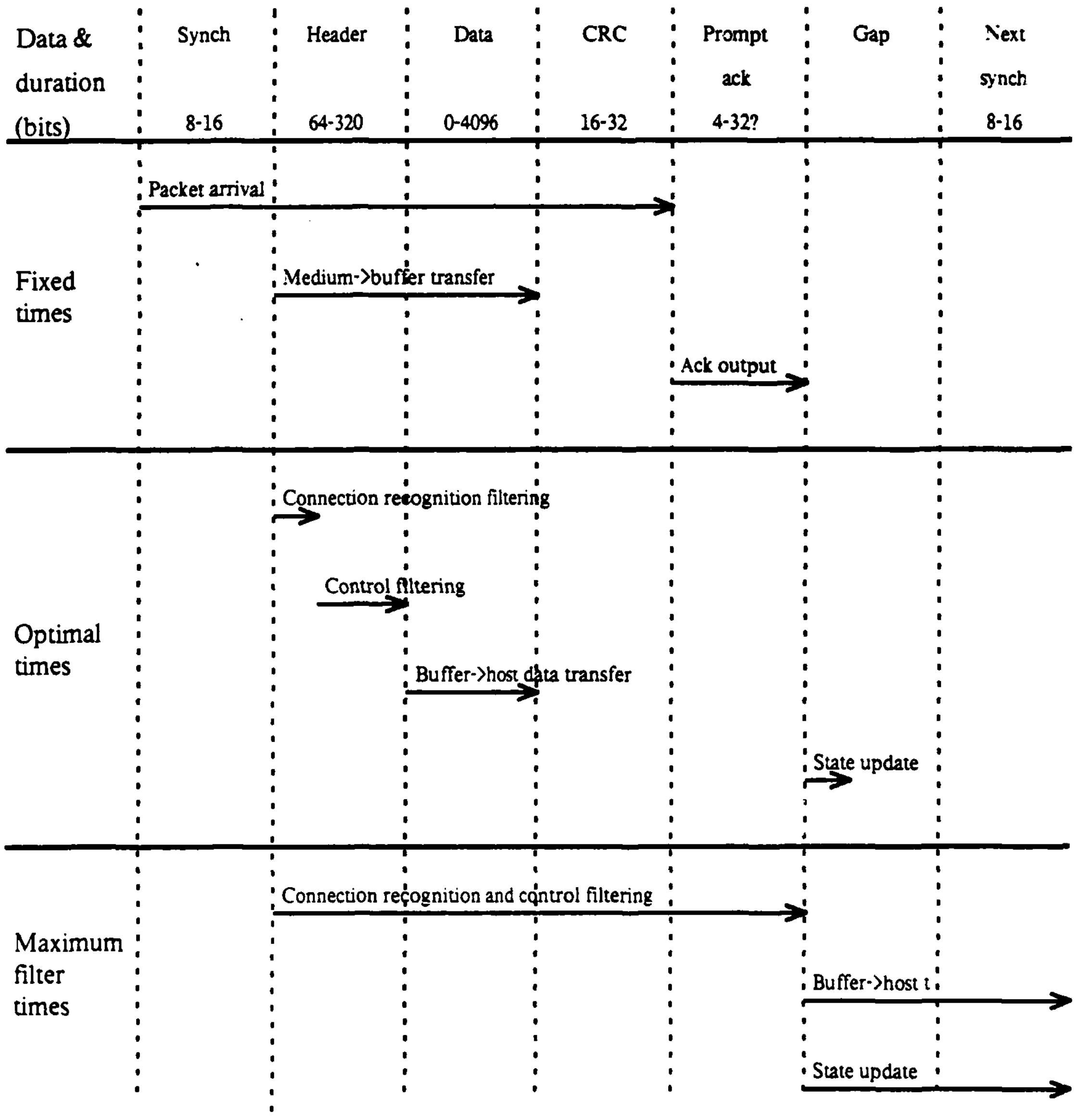


Figure 4-2: Model interface timing

The fixed timings and associated activities are strictly related to the signalling rate of the medium. Low-level functions, such as bit unstuffing, CRC calculation, and conversion between serial and parallel data formats are best performed by dedicated bit serial hardware. Converting the data flow to parallel format reduces the rate of operations; this parallelism is essential if programmable elements of the interface are to achieve an acceptable effective data rate. Given the available components, a parallel width of 8 or 16 bits is desirable. Because most protocols are based on the use of octets or 8-bit bytes, 8-bit data paths are preferred. The prompt acknowledgment output will probably require special hardware, given that the acknowledgment must be based on the correctness of the immediately preceding CRC.

The optimal timings assume that processing is performed as soon as the required data is available. Filter activities begin with a search of the filter database for the connection record which corresponds to the arriving packet. This search is based on the addresses in the packet, and may include identification of the protocol in use for interfaces supporting multiple simultaneous protocols. Once the connection record is located, the filter calculates the control signals and parameters for data-delivery tools. Figure 4-2 illustrates the activation of buffer-to-host data transfer. State update is performed only after the CRC and prompt acknowledgment have been seen to be correct. (Figure 4-2 assumes that the data-buffering discipline in the host allows the interface to put data in the host's buffers which the interface may later find to be invalid due to CRC failure or prompt acknowledgment problems.)

The optimal timings severely constrain the types of implementation structure that can be used to perform filtering. For example, assuming the 16-bit source and destination addresses used in the original RI, and the RI's 2 Mbps medium rate, optimal filtering must be accomplished in (16+16 bits)/2 Mbps = 16 usec. This amount of time is insufficient for any type of MOS microprocessor assist to the filtering function, even if the microprocessor can begin the filtering task 4 usec after the synch pattern alerts the interface to the arriving packet. The 16 usec interval may even be a problem for a bit-slice microprocessor, given that it corresponds to 80-160 microinstructions at best. Clearly, measures to relax the timing would result in more implementation flexibility or a larger connection space capability. The only cost to such a scheme is additional latency between data arrival and transfer to the host.

The maximal filter times assume that the results of filtering need only be available by the time the prompt acknowledgment is sent. Data transfer to the host is delayed until the interface has completed filtering; for short packets, the transfer is delayed until the prompt acknowledgment is output. State update can continue until the start of the next packet's header data. In the case of the RI example, filter times are doubled to 32 usec.

Filter times may be further extended by inserting delay between the header data passed to the filter and the prompt acknowledgment. One method is to mandate a larger minimum data length; this technique has the advantage of adding no overhead to packets which are already larger than the minimum. A second technique is to insert a fixed delay between the CRC and the prompt ack. Some minimal delay is already necessary in bus systems, such as the Hyperchannel, to change between bus transmitters.

Another constraint on filter design is the storage capacity for connection records, both in terms of the number of connection specifications (names) and the amount of specification data required per name.

The number of names required is highly variable, depending on host characteristics. A terminal on a local network needs only one or two names. A conventional timesharing host may need 100 simultaneous connections. Envisioned distributed processing systems will use an order of magnitude more connections. For a host that could use the performance of the model interface to advantage, somewhere between a dozen and several hundred connections seem reasonable. Therefore, an expandable filter with a minimum of 16-32 and a maximum of several hundred names seems tc be a reasonable choice.

Bounds on the amount of data per connection are less speculative. As a lower limit, the filter must recognize the addresses in the packet. Table 4-1 illustrates the address size, address components, and broadcast-address modes for several interface (hardware) and protocol (software) systems. The entries in this table are arranged in roughly chronological order.

The "size" columns give the total number of bits in an address; the "components" columns describe the size and function of the components of an address. The "broadcast" columns describe the components that can be universally quantified. For example, "* host" means that an address can have a host component which will match any host value. With the exception of the LNI, a broadcast component is encoded as a special value of the corresponding field.

This data supports the conclusion that address sizes are growing and vary considerably in length. For the model interface, a lower bound on the size of a connection ID is equal to 2 addresses of 64 bits, or 16 bytes. An upper bound on connection ID size can be derived from the header length of a packet; for an IP encapsulated TCP packet, this value is 40 bytes.

Filter design alternatives

In order to meet the timing constraints, the filter needs to be fairly fast, and must have deterministic worst-case performance to guarantee filtering is complete before the prompt ack time. Thus the design should be as simple as possible, and should use deterministic search algorithms.

The two parts of filtering, connection recognition and control filtering, seem to be different in nature, and hence may require different computational support. Connection recognition is a search problem: The filter searches the filter database for connection record(s) that match the packet's connection ID (usually the source and destination addresses). The control-filtering phase is more computational: The filter performs sequence space and other calculations that determine what actions, if any, should result.

Three filter organizations are possible:

1. A parallel search using associative memory similar to that in the LNI.
2. A path-following approach in which the filter database is conceptually similar to a finite state machine (FSM) description, and the packet header is used as the inputs to the FSM.
3. A hybrid approach, which uses a parallel search for locating the connection record, and a nonparallel element, such as a microprocessor, for control filtering.

Associative search

Associative memories similar to that used in the LNI are internally based on rapid sequential

Table 4-1: Addressing parameters

	Hardware Address			Software Address		
System	Size	Components	Broadcast	Size	Components	Broadcast
Ethernet(old)	8		*			
Ethernet(new)	48		*?			
PUP				48	8 network 8 host 32 socket	* host
DCS/RI	16	4 class 4 host 8 unique	* host	16	same	
LNI	32		by bit			
Batnet	none			16	8 network 8 host	* network
NBS	16					* host
TRW	8	256 addresses divided into: 63 point to point, 1 broadcast 192 "functional" multicast/allocation channels				
IP				40	8 network 24 host address 8 protocol	
TCP				16	16 port number	
LLL network				Address: 64	unspecified hierarchical components	
				Capability: <248	<152 password+UID 32 properties 64 address	

comparison of key data against possible matches stored in RAM. Parallelism in search is obtained by additional comparison units, by increasing the number of bits compared in a single comparison, or by a combination of both techniques. The typical organization of such an associative memory includes a central controller, a RAM for names, a result RAM for recording comparison status for the corresponding name RAM entry, and the actual comparison logic. If parallel compare units are desired the two RAMs and the comparison logic is replicated.

The name capacity of a RAM unit is limited either by the comparison rate (to the number that can be compared in the allowable time), or by RAM capacity. The comparison limit is proportional to the product of available comparison time and width of a single comparison and inversely proportional to the time required for a single compare. A typical scheme might use a byte-wide compare, a 100 ns compare time, and a time limit equal to two bit times on the medium. At 1 Mbps, such a system could hold 160 names; at 10 Mbps, 16 names.

Path-following search

The path-following approach views the incoming header as input to a FSM described in the filter database. The terminal states of FSM execution map to connection IDs. Assuming a deterministic FSM model, the difference between this approach and the associative-memory approach is that only one search is made per packet.

In the simplest form, the FSM has no intermediate states, the packet address is used as an address for a RAM, and the contents of the addressed RAM cell are the connection state. This was the approach used in the TRW interface [Blauman 79]. Using modern memories, the original RI name table could be implemented using a single 64K RAM. However, the longer addresses that are common in protocols require unreasonable amounts of memory.

A larger address space requires the addition of intermediate states to the FSM and a mechanism for representing the transition rules appropriate to a given state. In an octet-oriented system, the transitions related to address filtering can be represented as 256 way branches to successor states selected by an arriving address byte. Control filtering could be implemented as branches based on arithmetic calculations. Outputs from FSM transitions control-data delivery functions.

The implementation structure for such a system would consist of RAM for the transition tables and connection records and a controller to interpret the transition tables. The controller could be fairly simply implemented using ALU slices for comparison and calculation (in a high-performance version) or perhaps a microprocessor for media with low data rates. Because the filter has a relatively long time for processing each byte of header (compared to bit serial rates), multiple control outputs at a given point in the header could be handled as multiple transitions, each with a single control output.

The optimal design adjusts the width of the selector to compromise between memory requirements, speed, and compatibility with the rest of the interface:

- The width of the selector is the primary influence on filter memory size. In worst-case CID spaces, the filter FSM graph structure is the tree which maximizes the number of states per CID (leaf) by branching CID paths early in the FSM tree. In such a situation, the amount of filter memory (M) required to store C CIDs, each of N bits, using a selector of width S, can be approximated by

  $$M = \text{block\_count} * \text{block\_width} * \text{block\_height}$$

  where

  $$\text{block\_count} = CN/S$$

  $$\text{block\_width} = \log_2 (\text{block\_count}) = \log_2 (CN/S)$$

  $$\text{block\_height} = 2^S$$

- The 2**S cost factor predominates and approximates the amount of unused memory in paths during worst-case situations. Unfortunately, all addressing schemes in use today are close to worst case; hence the selector size should be minimized wherever possible.
- The speed required of the filtering system is inversely proportional to S. Larger values of S allow slower filter control and filter memory, and hence lower cost and higher memory density.

Two choices seem feasible: an 8-bit wide selector (which is in keeping with a byte-oriented design) and a 4-bit wide selector (which can be easily interfaced to the remaining byte-wide sections of the interface). A smaller selector is undesirable because it requires faster, and hence more expensive, RAMs for filter storage. At 10 Mbps, a 4-bit wide selector must generate filter decisions every 400 ns, whereas a 2-bit selector must operate in 200 ns. A 4-bit selector strategy could meet this time and allow for one arithmetic and one N-way branch filter instruction; it would be difficult to build a 2-bit system with the same flexibility at a comparable cost.

A worst-case example reveals the superiority of the 4-bit selector size. For example, a 64K x 8 memory and 8-bit selectors allows for 256 branch tables of 256 entries each. Ignoring the need for special addresses to signal comparison failure and space for the connection data, and assuming a 32-bit address size, the worst-case capacity for this system is 85 addresses. A 4-bit selector system with a similar capacity (82 addresses) could be implemented using a 8K x 9-bit memory. The storage for the 4-bit system is only (8K * 9)/(64K * 8), which equals 14 percent of the 8-bit selector system. Equivalent results apply for larger CIDs: For 64-bit CIDs, the 64K x 8 system with 8-bit selectors holds 36 CIDs, while the 8K system with 4-bit selectors holds 35 CIDs.

A 16-bit wide filter memory and a 4-bit selector seems the appropriate choice. The 16-bit width can allow either an 8- or 16-bit micro to conveniently address the filter memory, perhaps using byte addressing for microprocessor references. The 16-bit width will also allow for byte-oriented storage of control-filtering data as well as a large number of special codes for addressing-connection data blocks, etc. An acceptable number of CIDs can be stored using 16K x 1 or 2K x 8 RAMs, depending on need.

Hybrid approach

Neither the path-following FSM scheme nor the existing associative schemes are suitable for use in their pure forms. The associative memory scheme can't deal with multiple protocol layers or variable length comparisons, while the path following approach suffers from state explosion for real protocols.

The major changes required for an associative model interface are a facility for multiple, programmable comparison rules to the main controller and the ability to use different rules in each comparison cell. For example, assuming support for both IP and a local network-packet format, it is unlikely that the same address comparison rule could be used. The controller must examine the incoming packet, determine the appropriate comparison rule, and only then begin the associative search. The need for different rules for different associative cells is seen by considering the case where one cell holds an IP-only connection record and another holds a local format-only connection record.

Changes to make the path-following approach more usable are to include a facility for performing arithmetic (length calculations, sequence arithmetic, etc.) independent of the path-following mechanism and to support the construction of path subroutines.

Comparative analysis

The advantages of the associative scheme are the following:

1. The associative approach offers a natural method of supporting broadcast facilities, because all addresses are processed in parallel.
2. The cost of adding a binding function is the memory needed to describe it in the comparison-rule memory; no per connection cost is incurred.

The disadvantages of the associative scheme are the following:

1. The associative memory is fairly expensive in terms of hardware component count.
2. The associative memory structure is unlike that found elsewhere in the interface; there is little prospect of economies on the basis of shared hardware.
3. Maintaining the associative memory requires special addressing for the associative cells' contents.

The advantages of the path-following scheme are as follows:

1. Because the path-following approach follows a single path at a time, it minimizes the amount of computation which takes place per bit of arriving message. The advantage can be used to reduce the required logic speed or increase the amount of computation that can take place per bit of arriving message.
2. This scheme allows for a great deal of flexibility in resource (filter memory) allocation; the amount of resources consumed by a filtering specification is commensurate with the specification's complexity. Variable-length addresses, multiple protocols, variable encapsulations of a higher level protocol, etc., could all be handled.
3. The transition tables can be stored in a dual-ported RAM accessible by both the filter controller and the interface microprocessor. Updates to the filter database could be performed without additional hardware support.
4. Additional capacity can be added cheaply and simply by increasing the size of the RAM.

The drawbacks of the path-following scheme are as follows:

1. Broadcast names similar to those in the LNI require large amounts of table space. Any sort of control filtering with multiple recipients requires power equivalent to that of the associative scheme.
2. The number of transitions necessary to encode a set of addresses is not proportional to the number of addresses. For example, assume the FSM accepts 2 4-byte addresses, and the addresses differ in a single bit position. If the difference is in the first byte of the addresses, then 2 separate sets of 3 branch tables will be necessary to realize the FSM for a total of 7 branch tables of 256 entries each. If the difference is in the last byte, then the first 3 branch tables can be shared and only 5 branch tables are required.
3. Whenever a name is added or deleted, the interface microprocessor will have to spend a fair amount of time in updating the tables.

Given these arguments, the path-following approach is superior **in all cases**, if it can be demonstrated that:

1. The filter memory requirement can be made acceptable.
2. Broadcast addressing can be supported.
3. State update and computationally *oriented control filtering* can be integrated with the pure path-following approach.

The next section presents an outline of a filter machine designed to meet these objectives and motivates the initial choices for speed, operation width, etc. The next chapter uses the design to implement filter specifications for a particular set of protocols and defines the exact speeds and capabilities required.

Filter machine implementation outline

Figure 4-3 presents a block diagram of the filter machine. Its operation is explained below.

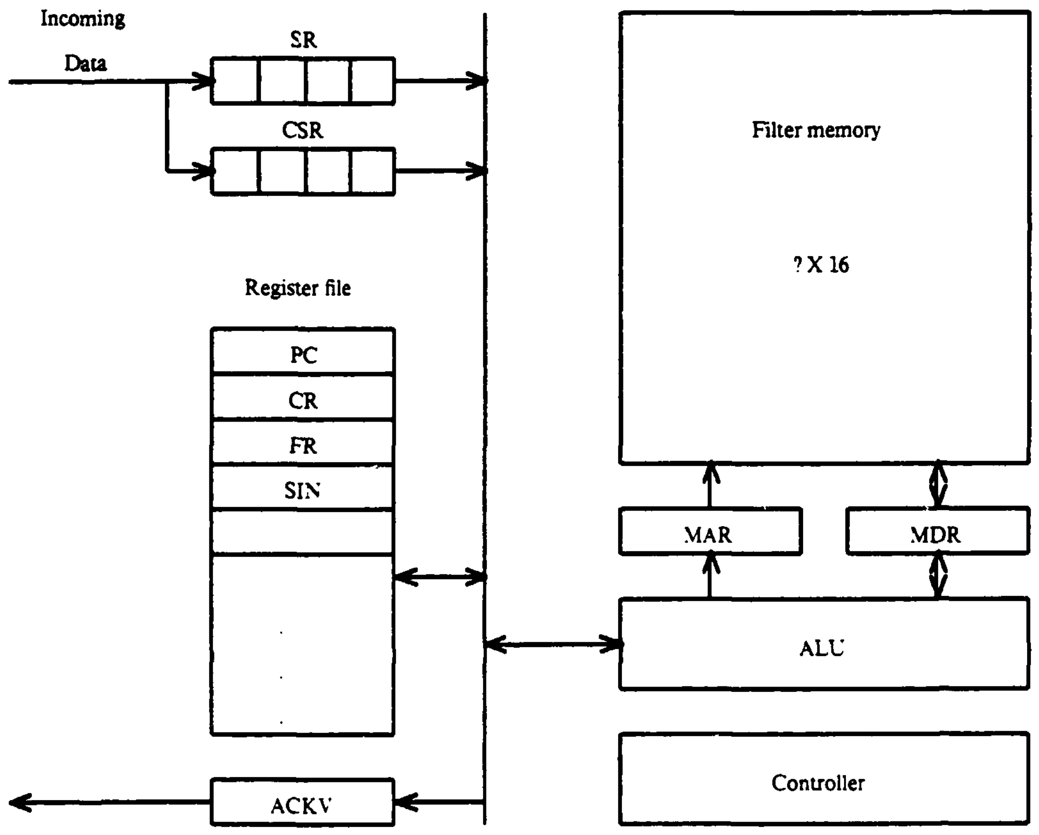


Figure 4-3: Filter machine block diagram

The filter memory is a 16-bit wide memory bank which is also part of the interface microprocessor's address space. Sharing and memory refresh (if required) could be implemented in a variety of ways, but the simplest method would seem to be to arbitrate access using a separate memory address register (MAR). If the filter machine is to process packets reliably, it must have priority to use the MAR. Refresh could be implemented in filter machine code or in interface microprocessor code if filter code sequences do not prove to lock out the interface microprocessor for extended periods. The interface microprocessor may regard this memory as being byte addressed, but word addresses are used by the filter.

The filter machine's instruction execution unit is of conventional design and includes a controller to govern filter instruction execution, an ALU section for arithmetic operations, and registers accessed through a common bus. Incoming data from the medium is buffered in the SR and CSR registers as described below; the results of filter machine execution take the form of changes to the filter memory (which are noticed by the interface microprocessor) and prompt acknowledgments which are serially output through the acknowledgment register (ACK).

The filter machine executes instructions that specify arithmetic operations, N-way branches, and memory operations. One instruction is executed for every 4 bits of incoming data. Thus for a 10 Mbps medium data rate, the filter machine has 400 ns per instruction. At most, a single instruction can generate 2 memory operations: one to perform a memory read or write specified by the instruction, and one to fetch the next instruction. At 10 Mbps, the memory cycle time needs to be 200 ns, well within the capabilities of current technology. The memory address is either directly represented in the instruction or is contained in the register set of the filter machine.

In addition to the optional memory operation, an instruction can perform 2 arithmetic operations. The operations are 16 bits wide. This size is adequate for counting down lengths and is found in many protocol formats.

Instruction execution has two phases. During phase 1, the memory operation specified by the instruction is performed, and one of the two possible ALU operations is performed. During phase 2, the instruction pointed to by the program counter register (PC) is fetched, and the other ALU operation is performed. Branch instructions modify the PC during phase 1. Memory accesses may implicitly require incrementing the memory address. For example, consecutive instruction execution increments the program counter (PC). Thus the maximum number of ALU operations per instruction is three, which can most easily be clocked by using the arriving bit clock and allowing 4 ALU operations per instruction. At 10 Mbps, this would require a 100 ns ALU cycle time. The first ALU cycle of each phase generates a memory address; the second ALU cycle in a phase performs a computation.

In order to encode all of the instruction parameters, an instruction width of 16 bits is required. Even this width may be inadequate for a straightforward encoding. This problem will be examined in the next chapter following derivation of actual code sequences. A two-level instruction cracking scheme may be required.

Data from the medium is made available to the filter machine in 4-bit chunks which arrive synchronously with respect to instruction execution. The arriving 4-bit chunks are shifted through a 16-bit register (SR), 4 bits at a time. The filter machine can access the 16-bit register to transfer 16 bits of consecutive data into the ALU as a single quantity. A separate 16-bit shift register (CSR) is provided and is shifted only under filter program control. The filter uses this register to extract and assemble fields from the incoming packet data.

The N-way branch instructions are executed using a table address stored in the filter machine's register set. At least two registers will be dedicated to this function: one register for connection recognition (CR), and one for format filtering (FR). The N-way branch instructions are executed using the following sequence of operations:

1. The base address of an N-way table stored in the CR or FR is combined with the selected predicate to form the address of an entry within an N-way branch table. The predicate is either the most recent 4 bits of incoming data from SR or 4 ALU status bits from the previous second ALU operation.
2. The contents of the addressed entry is fetched.
3. The addressed location contains two parts: a register designator and a value. The register designator selects the register which will receive the returning value. The designated register may be the PC, the CR, the FR, or a register yet to be designated.

The rationale for this design is that a set of N-way branch tables represents a FSM. The selected base register corresponds to the current state of the FSM. If a branch table entry contains a PC-directed value, it corresponds to a transition to a terminal state of the FSM. If the branch specifies a table base register, it corresponds to a transition to another state in the FSM.

4.5 REALIZATION

Alternatives for Interface Realization

The overall interface design contains several sections in which the choice of realization technology is obvious:

- The interface microprocessor and supporting memory should be built using conventional MOS microprocessor components. The reasons for this choice include cost factors, the availability of programming tools, and the need for a general-purpose computer to deal with the vast number of special case situations which occur in protocol processing. Some of the newer bipolar microprocessors and controllers could be used for increased speed, but their lack of instruction set generality and software support would greatly increase design cost.
- The transceiver and related hardware dealing with serial medium data rates must be implemented in digital logic of some form, rather than a microprocessor. The extent to which this logic is required is a function of medium speed and requirements for isolation and other electrical considerations. A slower rate can use digital phase locks and almost completely avoid the need for special line interface circuitry. However, analog phase locking and logic for serialization and deserialization will always be faster than a programmable implementation of any sort.
- The host interface will depend on individual host bus characteristics, etc.

The remaining question is the appropriate realization technology for the filter machine's instruction execution logic. The choices considered are bit slices, custom MOS, and a MOS microprocessor. The first two alternatives are preferred approaches for high-performance implementations.

Bit slices

This approach is especially natural for the processor described in the implementation section. For example, a few chips from the 2901 family could provide the majority of the filter machine logic. 2901s provide a 16-high register file, a shifter, and an ALU capable of addition, subtraction, shifting, and generation of zero, carry and borrow condition codes. Thus 4 2901 chips would implement the complete ALU, most of the required registers, and possibly the SR or CSR logic. External logic would be required for the PC (4 loadable 4-bit counters, such as the 74LS163), and miscellaneous registers for the MAR, MDR, condition code, etc.

The controller section of the filter engine would consist of PLAs or PROMs dedicated to generating the 2901 control signals from the instruction word, plus discrete logic or PLAs for filter memory arbitration, system initialization, etc.

2901s are available in several speed ranges, including a version that has the 100 ns ALU cycle time required for a 10 Mbps medium speed. Instruction execution has 4 phases:

1. The address of the memory operation is output to the MAR latches and a memory cycle is started.
2. The PC is incremented, memory data replaces the specified register, and the PC is sent to the MAR to begin instruction-fetch memory cycle.
3. ALU op 1 is performed.
4. ALU op 2 is performed, and memory data is latched into the instruction register.

The filter machine (excluding memory) is estimated to require a total of 15-30 ICs.

Custom MOS

Given the simple structure of the filter ALU controller and registers, and assuming outboard filter memory, the considerations for a custom MOS filter machine design are speed and connectivity.

The 100 ns time required for 10 Mbps processing is within the range of faster MOS processing abilities, particularly when all of the registers are on chip rather than offchip. The design allows for such a structure.

The connectivity problem is whether a custom chip could be built using an acceptable number of inputs and outputs. Exclusive of the filter memory interface, these signals are estimated as requiring 8 lines for incoming data, incoming clock, data present, reset, ACK register output, and power and ground. The address interface will require approximately 4 lines for control: read or write, ready, bus request, and a strobe. Assuming a 16-bit data path for the MDR, this totals 28 lines. A 40-pin IC could thus allow for a 12-bit or 4K filter memory address space. If additional addressing is required, encoding the control lines or multiplexing the address and data lines would still allow for a 40-pin package. Assuming a 64-pin package, no restrictions apply.

Thus a custom MOS implementation of the filter machine seems realistic for the 10 Mbps target. A higher rate might be possible if parallel ALU operations were incorporated in the design, and if the ALU rate, rather than the memory access rate, is the factor limiting performance.

MOS microprocessor

A microprocessor implementation of the filter machine would be possible, but would severely restrict the data rate. The extent of the speed penalty is related to the execution rate of the microprocessor and the fact that filter ALU operations can require several microprocessor instructions. For example, the 16-bit ones complement arithmetic required by IP/TCP checksums requires at least 2 and probably 3 instructions on a MOS twos complement machine. Assuming that all filter instructions, consisting of 2 ALU operations and a memory operation could be simulated as 3 microprocessor instructions per ALU operation and that 1 microprocessor instruction is required for the memory operation, a total of 6 microprocessor instructions are required per filter machine instruction simulated.

Assuming a 1 MIP microprocessor the filter data rate would be reduced to not more than 4 bits every 6 microseconds, or 666 Kbps. The actual throughput would probably be less, perhaps only 100 Kbps.

5. MESSAGE BINDING

5.1 OVERVIEW

The Scenario

This chapter covers the design of a filtering specification for an interface supporting both an IP/TCP protocol system and a local message protocol. Following this design, this chapter proposes extensions to filtering to allow broadcast service.

The protocol hierarchy for the filter is shown in Figure 5-1. This family of protocols follows the assumption that a local host will need to communicate via standard protocols (in this case IP and TCP) and a local protocol with simpler properties (LOCAL). LOCAL is a protocol similar to that used in the DCS RI and LNI systems; the changes are to enlarge the addresses and omit the mask fields of the LNI.

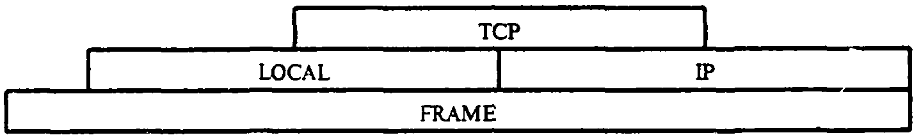


Figure 5-1: Protocol hierarchy

The format for the FRAME and LOCAL packets is shown in Figure 5-2.

8	8		8	16-32
Synch	Type	Encapsulated packet	Synch	CRC/checksum

Frame level packet format

48	48	16	
Dest	Source	Length	Data

LOCAL level packet format

Figure 5-2: FRAME and LOCAL packet formats

The FRAME protocol is similar to the message formats recognized by contemporary interfaces in that it contains a synch pattern to indicate the start of every message and a CRC or checksum field to detect transmission errors. All data except the synch pattern is bit stuffed; the choice of a particular CRC or checksum algorithm is deferred until the need for CRC or checksum support for other protocols is defined. The CRC or checksum is preceded by a synch marker to synchronize the CRC position and the subsequent prompt ack time period.

Unlike contemporary interfaces, the FRAME protocol contains no address or length fields; these functions are assumed to reside in the encapsulated protocol level(s). Omitting these fields from the FRAME-level avoids the need to duplicate length verification and addressing support; the encapsulated protocol may even suppress the FRAME CRC or checksum verification. For example, an IP packet carrying voice or facsimile data may only need to have its IP header checksum verified. The Type field in the FRAME protocol specifies the protocol layer immediately within the FRAME layer. In the scenario, the choices for Type are IP or LOCAL.

The LOCAL protocol is always encapsulated in a FRAME-level packet. LOCAL packets can be used as a top-level protocol for transfers within the local environment, or as an encapsulating layer for TCP segments within a local environment. The address size for LOCAL source and destination addresses must be at least 32 bits for compatibility with the TCP pseudoheader; a 48-bit length allows a set of distinct addresses for TCP encapsulation, a set of addresses for pure LOCAL use, and a large set of unused addresses for future expansion.

The LOCAL-level addresses in packets which encapsulate a TCP segment include 4 bits which signify an encapsulation, 8 bits which identify the encapsulated protocol (in this case, TCP), 4 bits of unused space, and 32 bits of internet address. Pure LOCAL packets can allocate the 44 bits after the first 4-bit field for any purpose; the allocation assumed herein is that 32-bit addresses are used for compatibility with the internet, and that the remaining 12 bits are used to carry sequence bits, MDF types similar to those described in chapter 3, etc.

The IP and TCP packets are identical to those defined in the relevant standards [Postel 81a, Postel 81b].

Figure 5-3 illustrates the various encapsulations possible with this system of protocols.

The filter designs for individual protocol layers must include connection recognition, control filtering, and prompt ack generation. Before considering the filter designs for each of these levels, we consider strategies for composing the individual filter specifications into a complete filter.

From a filter-specification designer's point of view, the best system is one in which filter specifications can be transparently nested for multiple protocol layers. One difficulty with this approach is that nested protocols require consideration of all levels of nesting rather than simply that of the deepest layer. In the case of IP and TCP, this problem manifests itself in the form of shared data in the pseudoheader and packet lengths from IP that must be used by TCP in computing the amount of data in a TCP segment.

A more realistic point of view is that nesting is handled by a combination of filtering code specific to each enclosing protocol and interfacing code which is specific to the interface between the two layers. A practical advantage to this approach is that the trailing data of one level can be processed concurrent to the arrival of the leading data of the next level. This is an advantage because processing of the first level's trailing data is often impossible until it all arrives, and hence the second level's incoming data has begun. However the next level has the same problem, and often cannot begin processing until a "critical mass" of second level protocol control has arrived. Overlapping processing makes use of the "wasted" filter cycles at the start of the second level to finish processing the first level.

Figure 5-4 is an outline of the complete filter FSM constructed according to this principle. Note that the states in this filter are actually sets of states rather than primitive states.

The starting state of the filter branches according to the FRAME-level TYPE field. One path out of

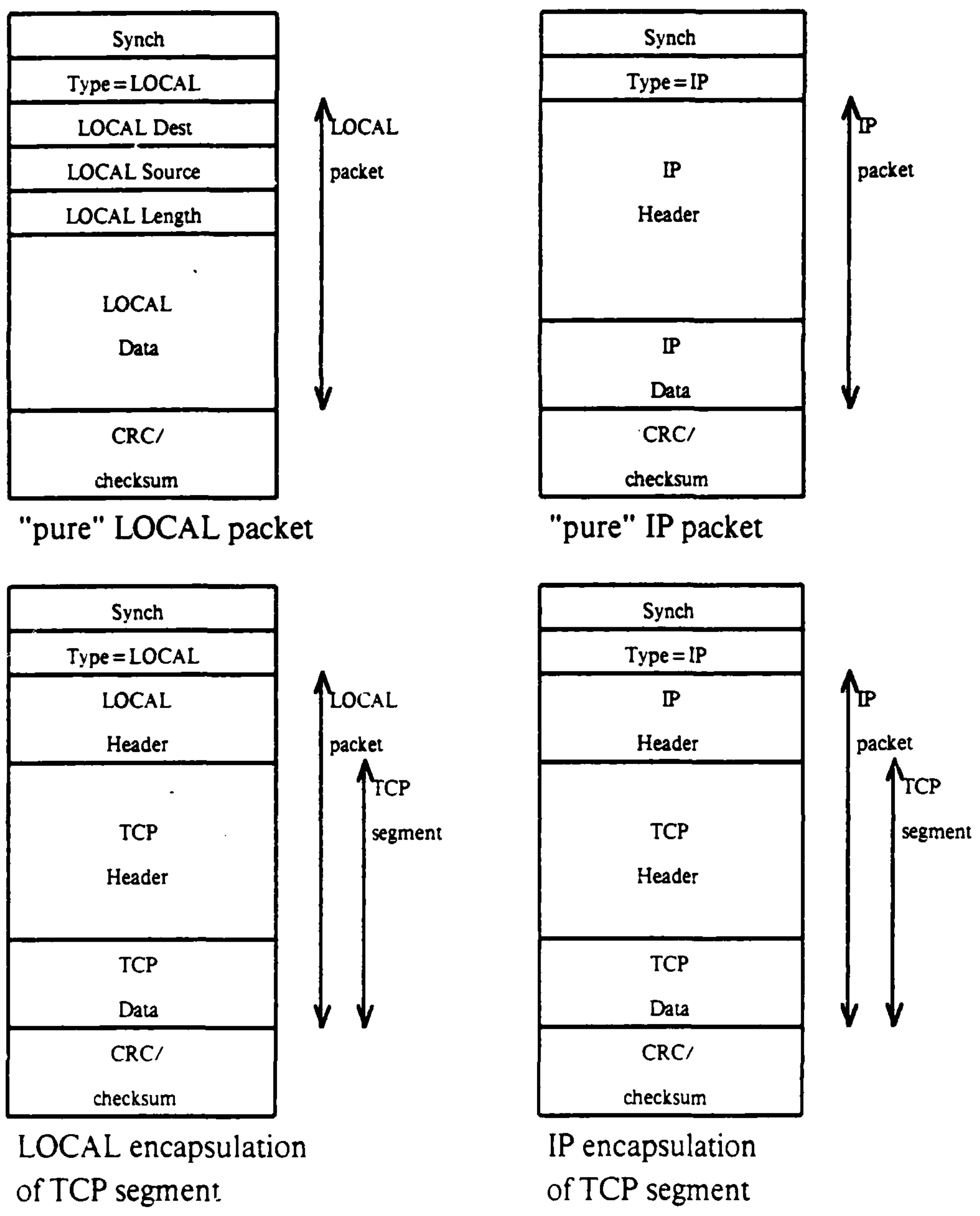


Figure 5-3: Possible packet encapsulations

this state corresponds to an unknown type field labeled "FRAME epilogue." The epilogue state waits for the synch preceding the CRC and then verifies the CRC. Depending on the filter design and the network medium, a prompt acknowledgment may be required. In the normal case, the start state branches to the "TCP encapsulation?" state of either IP or LOCAL after performing initialization for the next level. Given the small amount of time available, and the lack of information to direct initialization, the initialization is confined to setting up the N-way branch table pointers (CR & FR).

The "TCP encapsulation?" states test to see whether a TCP segment is encapsulated in this

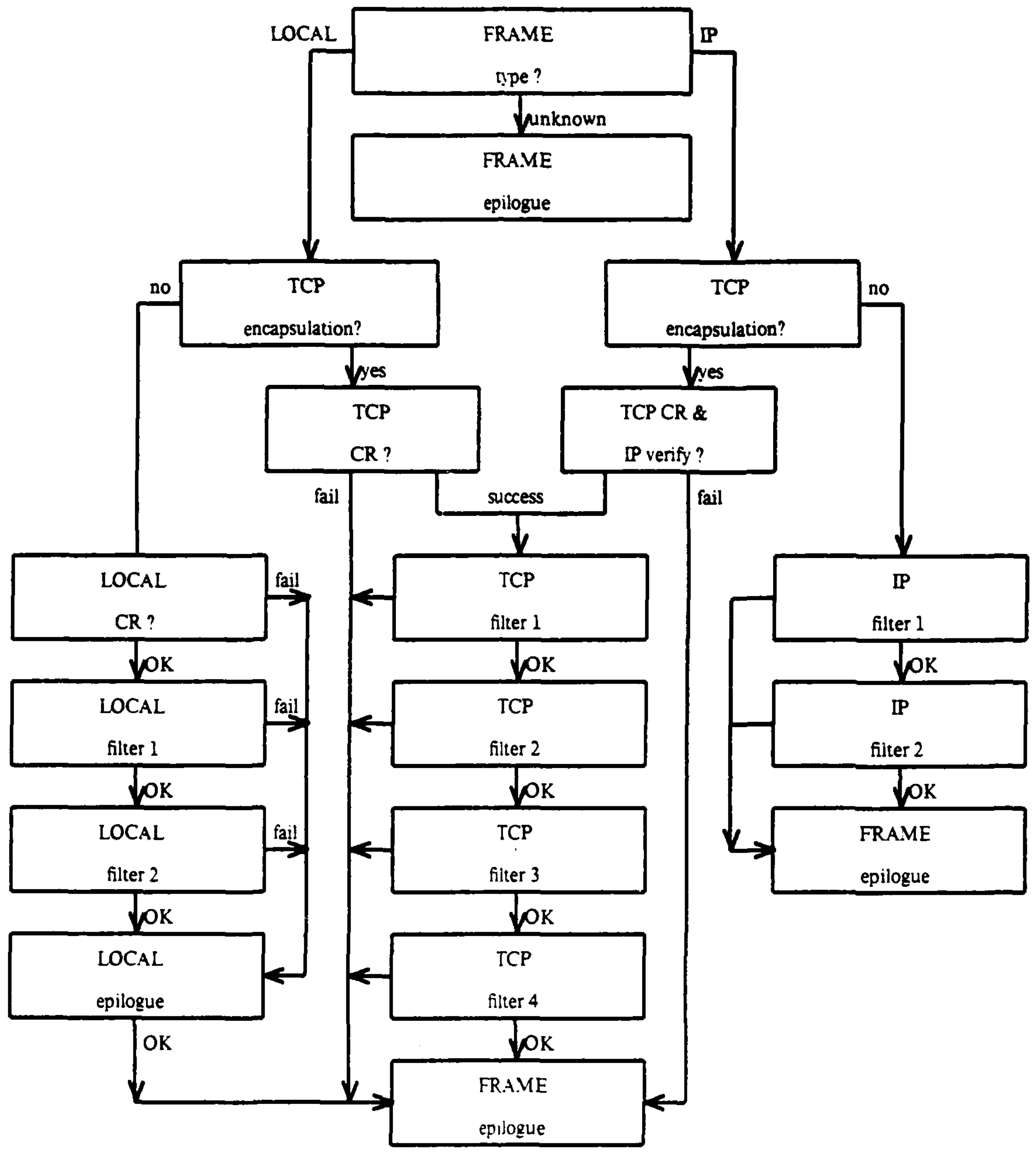


Figure 5-4: Filter FSM outline

packet. Two separate states are required to implement the different tests for IP and LOCAL. If the packet does not carry a TCP segment, the filter for pure IP or pure LOCAL is given control. If a TCP segment is present, the appropriate TCP filtering is begun.

The "TCP CR?" state is reached by a LOCAL packet which carries a TCP segment. This code contains elements pertaining to both the LOCAL and TCP layers. This state uses the LOCAL-level source and destination addresses to begin the search for a TCP connection record. If this lookup fails, a FRAME epilogue state is given control. While performing the lookup, the TCP segment length is set up, and the TCP checksum is set to the value appropriate for the TCP pseudoheader. Processing overlaps the boundary between LOCAL and TCP to complete the CID search and execute a conditional branch based on the IP checksum accuracy.

The "TCP CR & IP verify?" parallel the "TCP CR?" state code, except that the code is tailored to IP encapsulated TCP packets, rather than LOCAL encapsulation.

Following the initial states, three filtering programs handle pure IP, TCP, and pure LOCAL processing. These programs all attempt to perform sequences of filtering steps with the aim of performing as much filtering as possible. The filter program flows are likely to be more complicated than those used in Figure 5-4.

Format for Individual Protocol Descriptions

The filter definitions for the following individual protocols have four standard components:

1. *Ack development*, which defines the information the receiver would like to return to the sender of the packet. Note that the ack development sections are aimed at maximizing the amount of ack information, and that these acks may be suppressed for certain media or preempted by FRAME CRC errors etc.

   For example, an Ethernet implementation would be designed so that at most one ack could be generated, regardless of the number of filters on the system which generated ack conditions.

2. *Triage*, which defines the filter structure as it relates to separate cases of packet processing. For example, a filter might divide incoming packets into four classes based on whether the packets have a good checksum and whether they carry a recognizable source address.

   Triage also includes the normal action to be taken with the incoming packet's medium buffer. One choice is to discard the buffer because it has no further use. Another choice is to attempt to save the buffer for later processing by the interface microprocessor. This action is contingent on the availability of another buffer to accept the next packet (and an interface policy to do so). The interface should always be able to at least accept new packets which can be completely processed by the filter. Buffering strategies are discussed in the next chapter.

3. *Activities and code*, which describes the individual computational activities that must occur during packet processing, and uses pseudocode descriptions and execution traces to demonstrate practical filtering code sequences.

4. *Alternatives and extensions*, which describes alternatives and extensions that might also be implemented.

There are many tradeoffs between memory and processing, completeness and complexity, etc. The design presented is a system which has the ability to handle all packets, but which limits filtering to those packets which make up the majority of traffic. The filtering algorithms are designed to illustrate the minimum amount of concurrent processing which can perform the required tasks. This implies that their memory usage is often greater than that which could be achieved by a more complicated filter processor.

5.2 IP

Ack Development

Because IP is a datagram protocol, it has no acknowledgment structure inherited from the user interface; hence, the IP filter creates prompt acks for the purpose of increasing efficiency. Accordingly, only three fairly weak conditions are acknowledged:

1. *IP header checksum error*. This ack bit is only meaningful in the presence of a successful FRAME-level checksum or CRC check.
2. *Destination address recognized*. This ack bit signifies that the destination address is recognized. Although in some situations, notably gateways, the filter should recognize the addresses of several hosts or even a whole network, a single host is assumed for this discussion. The filter needs to recognize at least one fully qualified internet address as well as the corresponding addresses with a net field of zero. (Net = 0 signifies "this" network.) Additional pairs are required if multiple homing is supported.
3. *Protocol recognized*. This bit denotes that the protocol field of the IP packet corresponds to a server at this host.

Triage

IP filtering separates packets into several classes; unlike the ack bits these cases are mutually exclusive:

1. Packets with a bad IP header checksum are not processed further and are returned with the IP header checksum error ack bit set. These packets are discarded.
2. Packets with an unknown version field value are returned without ack bits. These packets are discarded.
3. Packets with IP options are not directly processed by this IP filter. Because IP options deal with security, routing, and satellite data, local packets typically won't carry options. The "Alternatives and Extensions" section discusses the processing of IP options. In this implementation, these packets are deferred.
4. Packets which are fragments are not processed beyond destination and protocol checks. If possible, these packets are deferred.
5. Packets which don't have a recognized destination address are ignored. These packets are discarded.
6. Whole packets with an unknown protocol are not processed beyond destination address verification. If possible, these packets are deferred.
7. Whole TCP segments whose source address is not found in the IP section of CID lookup. If possible, these packets are deferred.
8. Whole TCP segments with a recognized source address are passed on to the TCP filter which determines their disposition.

Activities and Code

The IP filter computes several values used to sort packets:

1. *The IP header checksum.* This value is verified by a ones complement addition of all 16-bit values in the IP header. The filter ignores the special case of a nonchecksummed IP packet signified by a negative zero checksum field, although this "feature" could be trivially added using N-way branches to detect this special case and duplicating filter code sections that demand a correct checksum.
2. *The TCP pseudoheader checksum.* This value is calculated for use by the TCP filter code.
3. *The length of the IP data block.* This value is passed to the TCP filter for use in sequence space calculation, etc.

A filter code execution sequence for IP is shown in Figure 5-5. This sequence corresponds to the execution for an IP-encapsulated TCP segment which is successfully passed to the TCP filter. Each line describes a single filter instruction for the corresponding four-bit input.

The first two columns relate the instruction being executed to the byte offset and contents of the arriving IP header data. The next eight columns describe specific ALU operations. "X"s indicate ALU operations which are fixed in position with respect to the incoming data stream; "*"s are ALU operations which need not be performed at a particular point. The "*" operations have been distributed to demonstrate that no more than two ALU operations need be performed per instruction. The last two columns describe the memory operation for the instruction and the N-way branch base register for those instructions which need these operations.

The IP filter uses three of the filter machine's GPRs to calculate the three required quantities.

- Register C1 is used to verify the IP header checksum. C1 is loaded from SR as soon as the first 16 bits of the incoming data stream are available; SR is added to C1 at 16-bit intervals thereafter. At the end of the sequence, C1 is zero if the IP header checksum was correct.
- Register C2 is used to accumulate the TCP pseudoheader checksum. C2 is initially loaded from CSR with a two-byte value which has zero in its high byte and the protocol field in its right byte. (The CSR value is computed using a trick which depends upon a zero fragment offset field. If this source of zeros for the CSR is not available, some other ALU operation is required, for example, CSR = 0.) Additional pseudoheader fields are added from SR for the source and destination addresses and the L register for the TCP length.
- The L register is used to calculate the length of the IP data segment which in this case is the length of the encapsulated TCP segment. The total length of the IP packet is loaded into L via CSR and decremented to subtract off the IP header length.

The N-way branches through the protocol check assume a set of N-way branch tables shown in Figure 5-6. During FRAME-level initialization, FR was set to point to T1. (Labels in this diagram are used for reference purposes only and are not used by the filter.) The arrows trace the sequence of N-way entries which are selected by a valid IP-encapsulated TCP packet.

	IP field:	C1= SR	C2= CSR	C1+ SR	C2+ SR	CSR EX	L= CSR	L= L-1	C2+ L	Memory operation	N reg
0	Version									Version=4	FR
	IHL									IHL=5	FR
1	Type of Service										
		X									
2	Total Length					X					
						X					
3						X					
				X		X					
4	ID						•				
								•			
5								•			
				X				•			
6	Flags							•		Flags=0*00	FR
	Fragment Offset					•		•		=0000	FR
7						•		•		=0000	FR
				X				•		=0000	FR
8	Time to Live							•			
								•			
9	Protocol					X		•		=TCP (high)	FR
				X		X				=TCP (low)	FR
10	Header Checksum		•					•			
								•			
11								•			
				X				•		CR = TCP CIDs	
12	Source Address							•		4 bit N-way	CR
								•			CR
13								•			CR
				X	X						CR
14								•			CR
								•			CR
15								•			CR
				X	X						CR
16	Dest Address										FR
											FR
17											FR
				X	X						FR
18											FR
											FR
19									•		FR
				X	X						FR

LEGEND:
C1 = IP header checksum register
C2 = TCP pseudo-header checksum register
L = IP data section length

Figure 5-5: IP worst-case filter-code sequence

The first N-way branch uses the IP version field as an index. If this value is four, the FR is loaded with the address of T2. Otherwise, the filter machine PC is loaded with the address of filter code to handle this exception (VFAIL), and the filter machine leaves the execution sequence described in Figure 5-5.

Similar filtering operations ensure that the IP header length is five (no options), that the IP packet is not a fragment, and that the protocol field is six (TCP)[Postel 81c].

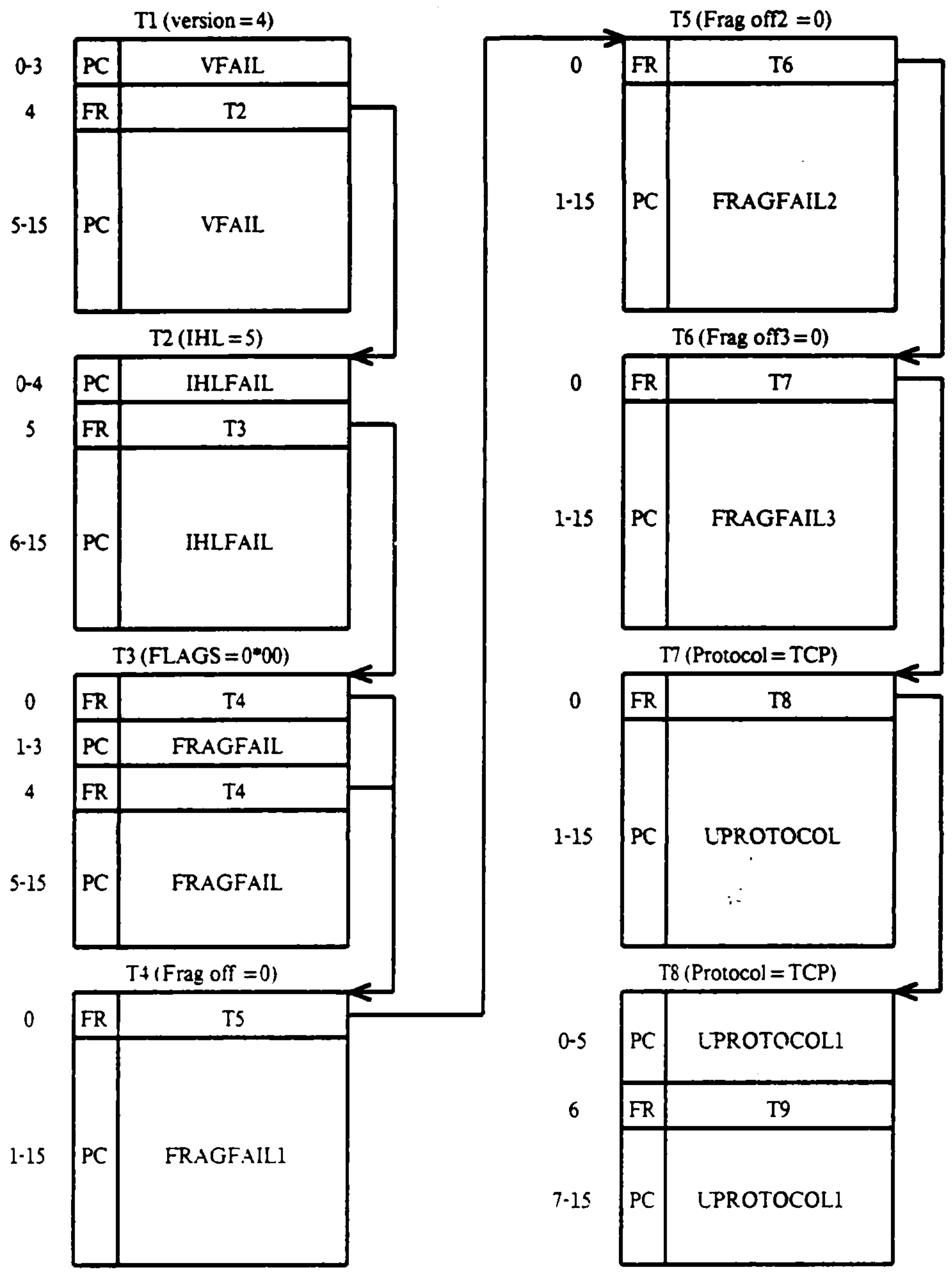


Figure 5-6: IP filter N-way branch tables

During the last nibble of the IP header checksum, the CR register is loaded with the address of the first N-way table that describes the CID space for TCP connections. The TCP CID space is a tree with much more branching than the FR-based search described in the example: the tree is 16 high and has as many leaves as there are active TCP connections. The 8 CR-based N-way branches partially search the CID space; a successful search is completed in the TCP filter code when port numbers are used to complete the path-following operation.

Following the source address search, the code reverts to FR-based branch tables to ensure that the destination address is for this host. (This assumes that the interface is receiving TCP segments for a single TCP server.) Using FR rather than a CR search for destination greatly reduces the number of tables necessary for TCP CID space recognition. However, if the destination can be referenced by more than one address, the different addresses don't enter into CID identification. In most cases this is a benefit rather than a penalty, but might not be the right choice in all situations.

Storage requirements

The IP filter needs storage for program and N-way tables. The storage requirements are summarized below:

1. Forty lines of mainline IP filter code plus an estimated total of 80 lines for other branches.
2. Eight N-way tables of 16 entries each for the filtering tables described in Figure 5-6.
3. Nine N-way tables of 16 entries each, and 1 32-entry N-way table for destination address recognition. Separate sets of tables are required for the mainline and other branches; hence this quantity must be doubled in the total.
4. Eight N-way tables per foreign host with TCP connections to this machine. (Worst case)

This yields a total of 400 locations for the IP filter and code plus an incremental cost of 128 locations per foreign host using TCP connections.

Alternatives and Extensions

The IP filter code discussed in this section is tailored to a network environment in which IP packets are most often used for TCP encapsulation, and hence does not support separate priorities, routing, fragmentation, IP options, or the efficient use of IP packets for datagrams. Support for these services requires a more powerful filter than that so far described for IP; however, the TCP filter described in the next section would be adequate.

Routing support

A number of filter code functions could be added to support routing and forwarding of packets, and hence build a high-performance gateway or store-and-forward switching node.

A very simple extension would be to use the IP address-recognition search to identify a queue for an outgoing link. Beyond this simple extension, the IP filter code could also decrement the time-to-live field and sort arriving packets according to the precedence, delay, throughput, and reliability values stored in the type of service (TOS) field.

When a gateway needs to send an IP packet to another gateway, it encapsulates the IP packet inside whatever format is applicable for the link which will carry the packet to the next gateway. For example, one gateway on the ARPANET sends IP packets to another gateway on the ARPANET by encapsulating the IP packet in an ARPANET format message. This outer layer carries the address of the destination gateway, rather than that specified in the IP packet. This strategy is used to limit the number of addresses that must be known on the link that carries the encapsulated IP packet.

Two strategies can be used in the filter to support traffic between gateways.

- The first strategy is to program the filter to understand all possible network addresses and hence avoid the need for the encapsulating layer. Each gateway's filter code would copy all IP packets addressed to networks that the gateway knows how to reach. One drawback of this scheme is that it is limited to situations in which the participating gateways all have filter interfaces. The second difficulty is that the sending gateway cannot select among alternate outgoing gateways; destination gateways control acceptance through their filter code. If two outgoing gateways both recognize and forward the packet, the packet will be duplicated, potentially resulting in an avalanche of further duplications. Solutions to this problem require coordination of gateways to avoid duplicate reception; this coordination must deal with failed gateways and links, and hence would have to dynamically reconfigure filter tables.
- The second strategy is to use IP as an encapsulating layer for IP packets between gateways. The outer IP layer addresses a destination gateway, and the inner IP packet addresses the desired destination. In this case, the filter could be programmed to remove the outer layer and queue only the inner IP packet.

Fragmentation support

Fragmentation of an IP packet is necessary when an IP packet must be transmitted on a network which has a maximum packet size which is less than the size of the IP packet to be transmitted (plus envelope, if applicable).

Within or across a single network, fragmentation can be performed by the local protocol used to carry IP packets; this process is invisible to the IP level and is called intranet fragmentation. Because the envelope protocol typically changes at network boundaries, intranet fragmentation is limited to a single network. Intranet fragmentation is found in several existing networks (e.g., the ARPANET). Local networks usually avoid the need for intranet fragmentation by limiting IP packet size.

A more general form of fragmentation is defined within IP which allows for internet fragmentation. In internet fragmentation, any node can fragment an IP packet or fragment into two or more fragments. Because fragments may take different paths toward the eventual destination, it may not be possible to reassemble the fragments until the final destination.

The internet fragmentation procedure uses the IP Identification field to identify fragments from a particular original packet. The source of IP packets uses the identification field much like a sequence number to identify separate packets. The fragment-offset field identifies the position of a fragment within the original packet; the last fragment is identified by the Last Fragment bit in the IP header. The typical reassembly procedure allocates a reassembly buffer when the first fragment arrives, updates the reassembly buffer as new fragments arrive, and output the packet when it is complete. Because IP fragmentation uses eight octet units, the reassembly buffer can be used to mark missing fragments in place. Unfortunately, the size of the original IP packet, and hence the size of the reassembly buffer, isn't known until the last fragment arrives; most reassembly algorithms deal with this by always allocating a maximum size reassembly buffer.

In practice, the complexity of the required reassembly and buffer allocation policies results in little use of this feature; most hosts don't implement reassembly, set the Don't Fragment bit in the IP header, and use packets of less than the IP minimum size (576 octets). However, the need for fragmentation support may grow as new networks are added to the Internet.

The filter could support fragmentation by performing the reassembly process, given a reassembly buffer and the arriving packet. Because of the memory cost of a maximum size buffer, the interface microprocessor should probably make the decisions to attempt reassembly or to abandon a reassembly effort.

IP options

As defined at present, there are several general categories of IP options. Depending on the environment, the IP filter code could be made to process one or more of these types to enhance performance. Parsing the options string to extract individual options is a fairly simple task; all options start with a single octet option identifier, and the most common types are fixed in length.

The simplest options are one octet long and signify the end of the option string or a no-operation. These options can be processed trivially.

Security and stream identifiers are two fixed-length option strings that specify packet security and the stream ID of the packet. Both of these options can be used to sort packets into queues, or to restrict the flow of packets, or otherwise control packet processing. The filter code for these options would resemble that used for address filtering.

The "timestamp" and "record route" options are more complicated for the filter to process, because they entail storing data into the packet and checksum modification. In both cases, these options specify a variable length data area in the option string which is used as an audit trail for the packet as it passes through nodes.

The record route option defines an area in which nodes that process the packet add their network address. The area includes a length field and a pointer which indicates how much of the area has already been used by previous nodes. Each node first examines the length and pointer fields to see if the area is full; if full, no further action is taken. If the area is not full, the node adds its internet address to the area, increments the pointer, and modifies the checksum. This processesing could be fairly simply added to the filter.

The timestamp option stores internet address and timestamp pairs instead of simple addresses, and uses an overflow counter to keep track of the count of entries that could not be recorded. In addition, the originator can fill in the internet address components, while leaving the timestamp components set to zero. In this case, the forwarding nodes must search for their address before storing a pair in the unused space at the end of the list. This more complicated procedure probably means that any filter support will delay processing beyond the arrival time of the timestamp area.

The source-routing IP options are the last class of IP options. The "loose source and record" option and the "strict source and record" option both specify a list of gateways which the packet traverses on its way from its source to its ultimate destination; the difference between the two is that the strict route specifies the exact sequence of gateways to be traversed, whereas intermediate gateways are allowed to process the packet between the gateways in a loose source list. In either case, the option consists of a type code, a length, a pointer, and a list of IP addresses. When a packet with either type of route reaches the destination address specified in the IP header, and the pointer is not greater than the length, the next address in the source route and the packet's destination address are swapped, and the pointer is increased by four.

Adding filter support for source routes requires that the header be rewritten, but does not tax the filter's processing power because the packet is either at its final destination, in which case little processing is required, or the packet is to be forwarded, in which case much of the normal IP processing and any processing for the data (e.g., TCP filtering) can be omitted. The only difficulty is that destination address filtering should be restarted after the destination and source rote entry are swapped.

5.3 TCP

Connection State Representation

In order to define filter processing for TCP packets which refer to an existing connection, the filter requires a state representation and a method for updating state in response to packet arrival.

Designing a state-control block is fairly simple; a contiguous block of memory locations is used to hold the values required by the filter in the order in which they are used.

The major problem in designing a state-update procedure is that all state updates (and associated side effects) are contingent on a correct TCP checksum, but the accuracy of the checksum cannot be known until the last octet of the TCP segment arrives. In order to allow processing in parallel with data arrival, the filter constructs a new state-control block as it processes the incoming data. Data delivery which overlaps previously acknowledged data is suppressed. State update is accomplished using a single pointer update which replaces the old state-control block with the new one. (The pointer to update is the last level of the CID recognition tree.)

The state representation and control block linking are illustrated in Figure 5-7.

Two state blocks are allocated to every existing connection. One holds the current state values; the other is used to accumulate a possible new state until the packet checksum can be validated.

The RCVnb field holds the address of the next state block. RCVleft holds the sequence number of the first octet which has not yet been accepted. RCVwindow is the size of the accepting window. RCVbufad is the buffer address for the octet referred to by RCVleft. RCVrack is the highest acknowledged sequence number for the reverse data channel. RCVrwindow is the sequence number of the right edge of the reverse channel window. RCVurgent is the sequence of most recent urgent.

RCVstate is a bit vector which describes the update history of the state values. The filter sets these bits; they are cleared by the interface microprocessor as part of the data-delivery function. The following bits are defined:

1. *NEWDATA* set whenever a new data octet is accepted.
2. *NEWCHOKE* set whenever a zero window is signalled in a prompt ack.
3. *NEWPSH* set whenever an octet marked with PSH is accepted.
4. *NEWACK* set whenever the ack value for the reverse channel is superceded.
5. *NEWWIND* set whenever the window value for the reverse channel is superceded.

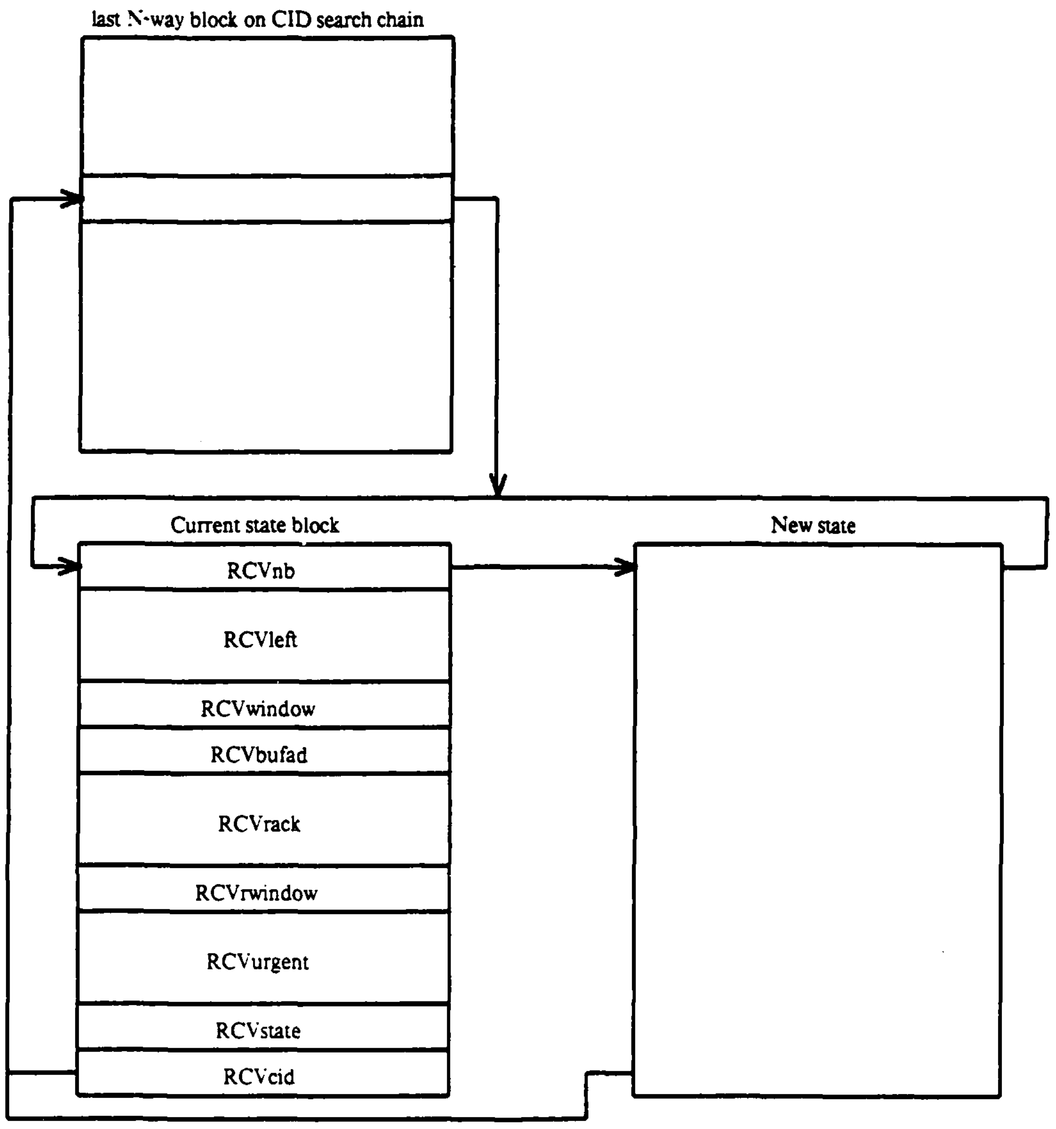


Figure 5-7: TCP state representation and control block linkage

6. *UPEND* set if an urgent value is present in RCVurg.
7. *NEWURG* set whenever the urgent value is superceded.

These values are an exhaustive set of events which require interface microprocessor intervention or activate data-delivery tools; strategies for the interface microprocessor are discussed in the next chapter.

Ack Development

The TCP prompt ack bits are as follows:

1. *TCP checksum error.* This bit signals a TCP checksum error and implies that all other TCP prompt acknowledgment bits are to be disregarded.

2. *Connection not found.* No connection exists for the specified port pair. This is the normal response to all SYN-bearing segments.
3. *TCP processing deferred.* This bit signifies that the TCP segment could not be directly processed by the filter. If this bit is not set, the assumption is that the acknowledgment, window, and checksum data were processed.

   This condition is caused by the presence of SYN, FIN, or RST flags, a sequence value which must be validated by interface microprocessor software, or the presence of TCP options. All of these conditions are either rare or too complicated to be processed by the filter.
4. *Control value ACK (CACK).* This bit specifies that the control fields in the segment which do not relate to data octets have been processed. These fields are the ack, window, and urgent fields.
5. *Acknowledge all data (ACKALL).* This bit specifies that all octets in the range between the sequence number in the arriving packet and the last octet carried by the segment have been accepted by the interface. If the packet carries no data, this bit acknowledges the single octet specified by sequence. Note that this bit guarantees only a lower bound and not a precise value.
6. *Receive window now closed (CHOKE).* This bit is set if the receiver can accept no octets.

Triage

The filter code for TCP separates processing into the following classes of arriving segments:

1. Packets with a bad TCP checksum result in a checksum error prompt ack and no state update. The packet buffer is discarded.
2. Segments addressed to an unknown connection generate a prompt ack which always includes the connection-not-found bit. If the buffer can be held for later processing by the interface microprocessor, processing deferred is also set.
3. Segments addressed to existing connections which contain options, SYN, RST, FIN, or strange sequence numbers (described in next section) are returned with a processing-deferred-prompt ack if the segment can be buffered, and a null-prompt ack otherwise.
4. The remaining class consists of segments which the filter can process itself; hence the segment buffer need only be retained until state update and data transfer is complete. Depending on contents and connection state, these segments return a combination of CACK, ACKALL, and CHOKE as prompt acknowledgments.

Activities and Code

TCP processing includes activities similar to those in the IP filter for locating the connection block, filtering out packets with unprocessable control fields, and checksum computation. The major new elements of TCP processing are new state construction, computation of values to direct data transfer, and a much greater amount of conditional processing. The major tasks are

1. Data octet processing, which decides which data octets from the segment are transferred from the arriving segment buffer into the host or interface buffers, as well as the associated state update and prompt ack calculations.

2. Reverse channel acknowledgment and window processing, which is conditional on the presence of an acknowledgment field in the arriving segment.
3. Urgent processing, which is conditional on the presence of an urgent offset in the arriving segment.

A major question in all of these functions is the design of sequence-space calculation policies. The TCP specification defines a ring of 32-bit sequence numbers for all sequence values, but doesn't describe a policy for comparing these numbers in all cases. During normal processing, all comparisons are between numbers which differ by a small number of octets (say 10-1000); hence the ordering relationship is clear. However, any implementation has to formulate a policy for dealing with sequence values which appear too "old" or "young" to be reasonable, even ignoring the issue of deciding between too old and too young. (Erroneous implementations, grossly delayed packets, and packets from previous connections are the usually quoted sources of these packets.)

This filter implementation sidesteps these issues by only processing segments whose sequence-space values (data sequence, reverse ack, urgent, etc.) are within 64K of "expected" values. This restriction is implemented by deferring the processing of segments when sequence-space-difference calculations result in a difference which cannot be expressed in the 16-bit result from a single ALU operation. To speed this check, sequence-space-differencing operations are always performed so that the most work is entailed by a positive value, which can be easily checked as a zero result for the high-order bits. Sequence space calculations are also simplified by the restriction that SYN- and FIN-bearing segments are deferred; this means that adjustments to sequence values are never required.

Pseudocode for the sequence space and data filtering is shown in Figure 5-8, acknowledgment and window code is shown in Figure 5-9, and urgent code is shown in Figure 5-10. This ordering is implied by the order in which operands become available; however, in the actual filter code, these operations are often reordered to balance the ALU and memory-operation load. Several notational devices are used in the pseudocode to illustrate actions which are difficult to express. Transfers to and from the state records are indicated by **input** and **output** statements. Conditional control is shown using if and case statements which are replaced by N-way branches in the actual filter code.

The sequence and data logic begins by calculating the difference between the left edge of the connection's data window and the sequence number of the packet in the variable `skip`. If `skip` is negative, the data octets in the segment are ahead of octets which have not yet been received; in this implementation, the data octets are ignored in the knowledge that they will be retransmitted. If `skip` is positive but greater than 64K, the segment's sequence value is deemed strange, and the segment is deferred for processing by the interface microprocessor.

In the normal case, `skip` is zero or a small positive number. `Skip` then represents the number of data octets in the segment which have previously been received. The filter goes on to calculate `toxfer`, which is the number of new octets in the packet. `Toxfer` is calculated using `tcpdlen`, which is the number of data octets in the segment, and `window`, which is the width of the connection's data window. (`Tcpdlen` is passed down from the IP filter.)

`Toxfer` and `skip` are used to control data transfer. Data transfer is suppressed for `skip` octets and then enabled for `toxfer` octets. Suppression insures that data octets from a segment with a bad checksum don't overwrite previously received and acknowledged good values. Writing the `toxfer`

```
input    sout=RCVnb             { pointer to next state block        }
         skip=RCVleft           { sequence of first desirable octet }
         window=RCVwindow       { maximum octets to accept           }
         bufad=RCVbufad         { buffer address for next octet      }
         rack=RCVrack           { reverse channel ack sequence       }
         rwindow=RCVrwindow     { reverse channel window sequence    }
         urgp=RCVurgent         { urgent pointer                     }
         state=RCVstate         { connection state                   }
         cid=RCVcid;            { pointer to CID master pointer      }
left:=skip;
skip:=skip-seq;
if high(skip)=0  {test to see if overlap possible}
then begin     {packet sequence may overlap RCVleft }
     toxfer:=tcpdlen-skip;   { calculate octets within window }
     if toxfer <= 0          { no data to transfer }
     then begin
          toxfer:=0;         { insure zero transfer count }
          ack:=ack+ACKALL;
          if window=0
          then ack:=ack+CHOKE
          end
     else begin     {new octets available, check window }
          case (toxfer = window)
          =:   begin
               ack:=ack+ACKALL+CHOKE;
               if PSH in SEGMENT.flags
               then state:=state+NEWDATA+NEWCHOKE+NEWPSH
               else state:=state+NEWDATA+NEWCHOKE
               end;
          >:   begin     { too many octets, truncate transfer }
               ack:=ack+CHOKE;
               if window=0
               then state:=state+NEWCHOKE
               else state:=state+NEWDATA+NEWCHOKE;
               toxfer:=window
               end;
          <:   begin
               ack:=ack+ACKALL;
               if PSH in SEGMENT.flags then state:=state+NEWPSH;
               end
          end; {case}
          end
     output left+toxfer,wind-toxfer,bufad+toxfer
     end
else if high(skip)=-1
     then output left,wind,bufad
     else strange_sequence()
```

Figure 5-8: TCP sequence and data pseudocode

octets into the connection buffer isn't a problem because these octets won't be acknowledged, or the window advanced, unless the checksum is correct. The only penalty for this activity is wasted memory cycles.

While calculating the data-transfer parameters, the filter updates the state and ack variables in accordance with data-transfer decisions.

```
{ if ack present, test ack & reverse window update }
if ACK_PRESENT in SEGMENT_flags
then case (ackseq:rack)
     >:   begin {new ack value to store}
          state:=state+NEWACK+NEWWIND;
          output ackseq,wdelta
          end;
     =:   if wdelta>rwind {test for larger window}
          then begin
               state:=state+NEWWIND;
               output rack,wdelta
               end
          else output rack,wind
     <:   output rack,wind
     end {case}
```

Figure 5-9: TCP acknowledgment and window pseudocode

The ack and window calculations are contingent on the presence of an ack flag bit in the flags field of the segment. If the ack bit is set, the acknowledgment in the segment is compared against the acknowledgment in the state record. If the new acknowledgment value is later, it replaces the old value. A new acknowledgment implicitly signals a new window allocation; the window value in the segment is otherwise ignored unless it is greater than that in the state record and the acknowledgments are equal.

The urgent logic is similar to that for ack in that a newer urgent sequence replaces an older one. Urgent processing is complicated by the necessity to verify that an older value is available; if no older value is available, the value in the segment is set in the state record.

TCP segments require more processing than IP packets and, unlike LOCAL, TCP cannot be redefined to fit the capabilities of the filter machine. Hence translating TCP pseudocode into actual filter code is a much more difficult problem than the analogous translation for IP or LOCAL. The close match between the capabilities of the filter machine and the demands of TCP processing requires careful compromises between performance, resource usage, and program complexity.

TCP segments cannot be entirely processed as they arrive. Urgent processing, final checksum verification, and state output must wait for the end of the arriving segment before required calculations can be performed; when no data is present in the segment, these calculations take place after the arrival period is complete. This extra time is maximized in a segment without data because of the small amount of processing performed on actual data octets. The amount of extra time must allow for at least 5 memory operations (2 urgent sequence output, 1 state output, and one pointer

```
if URG and ACK in SEGMENT_flags { URGENT value in incoming segment ? }
then begin
     seq:=seq+udelta;   { calculate sequence }
     if UPEND in state  { if old urgent, test for greater value }
     then if seq > urgp
          then begin
               state:=state+NEWURGENT;
               output seq
               end
          else output urgp
     else begin { no old urgent, believe new one }
          output seq;
          state:=state+NEWURGENT+UPEND
          end
else output urgp;
```

Figure 5-10: TCP urgent pseudocode

update), or approximately 3 bytes of time. A constant overhead of 4 bytes is assumed in the following discussion.

Using this assumption, TCP execution sequences will always have at least 48 filter instructions for processing the TCP segment header and up to 4 octets of data. The filter code design must allocate several "resources" associated with these 48 instructions including memory operations (loads, stores, N-way branches), ALU operations and registers, as well as the memory space for the filter program, N-way tables for filtering and connection recognition, and connection-state data.

Memory operations

The 48 instructions for any TCP filter code sequence must include the following:

- 8 N-way branches for port connection recognition
- 12 loads for state output
- 3 N-way branches for segment flags and data-offset filtering
- 9 N-way branches (maximum) for other filtering conditionals
- 11 stores for state output
- 43 Total

Thus approximately 90 percent of the available 48 memory operations are used for predetermined purposes, and only 6 memory operations are available for loading constants and filter program branches (which often require 2 memory operations: one for FR, and one for the PC). In addition, the placement of these operations is very constrained with respect to position in the filter code sequence and the ordering of the required operations.

Filter program structure

The scarcity of free memory operations influences program structure because of the difficulty in merging separate sequences generated by conditionals. We wish to avoid the case in which the filter code resembles a tree and no code beyond the "root" is shared because of the huge memory requirement this would entail.

Merging of alternate sequences is most effective in the position just before the segment flags are processed. This location merges all of the code sequences generated by the sequence and data-octet conditionals prior to separation due to segment flags. This ideal is unrealizable because some data-octet calculations cannot complete before the PSH flag is available. The best compromise is to merge into two sequences: one which is PSH sensitive (i.e., PSH implies NEWPSH) and one which is PSH insensitive. The PSH sensitive and insensitive code sequences can also be merged following the first four bits of the segment window field.

The major states of the two parts of TCP filter code are shown in Figures 5-11 and 5-12. The former figure shows the main states up to the merge point preceding segment flags. The latter figure shows the program structure following the conditionals for segment flags. The paths shown in bold face are the worst-case paths in which memory operations are densest, and hence the most difficulty is encountered in coding. In the latter diagram, the code above the dotted line is duplicated for PSH sensitive and PSH insensitive sequences.

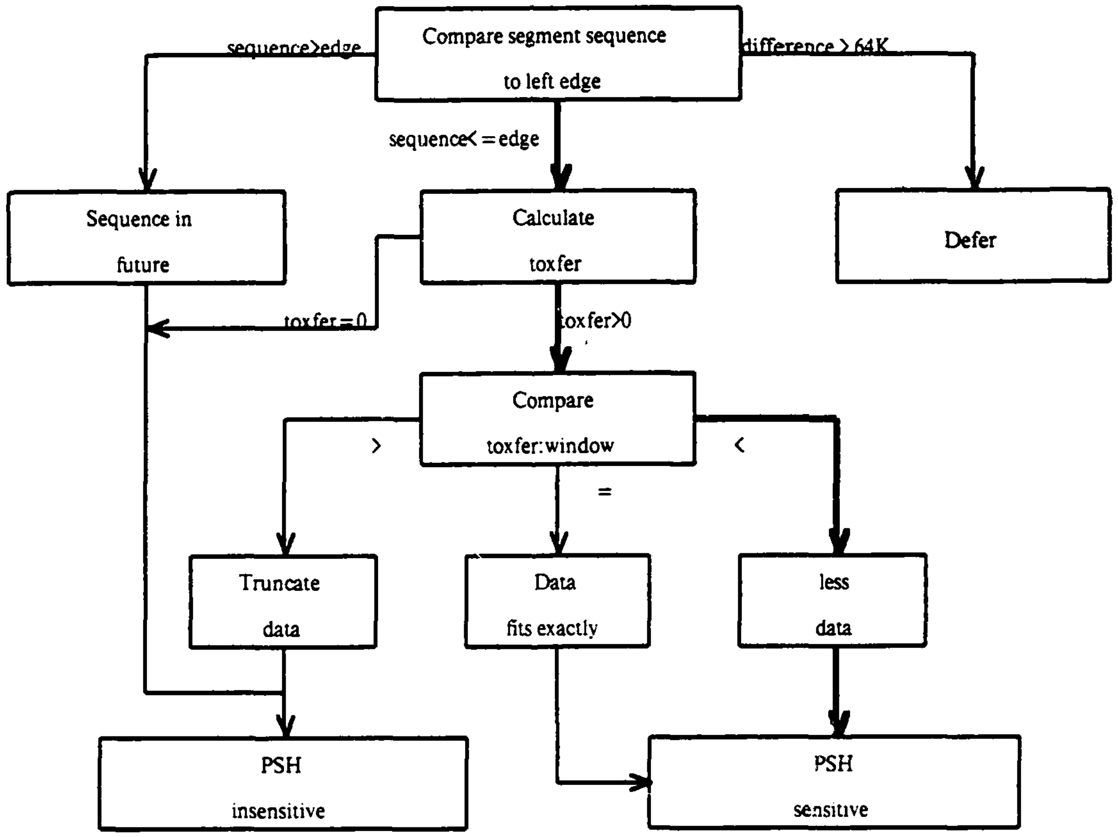


Figure 5-11: TCP filter code major states (part 1)

ALU operations

The 96 ALU operations contained in the 48 filter instructions are the only resource which is not

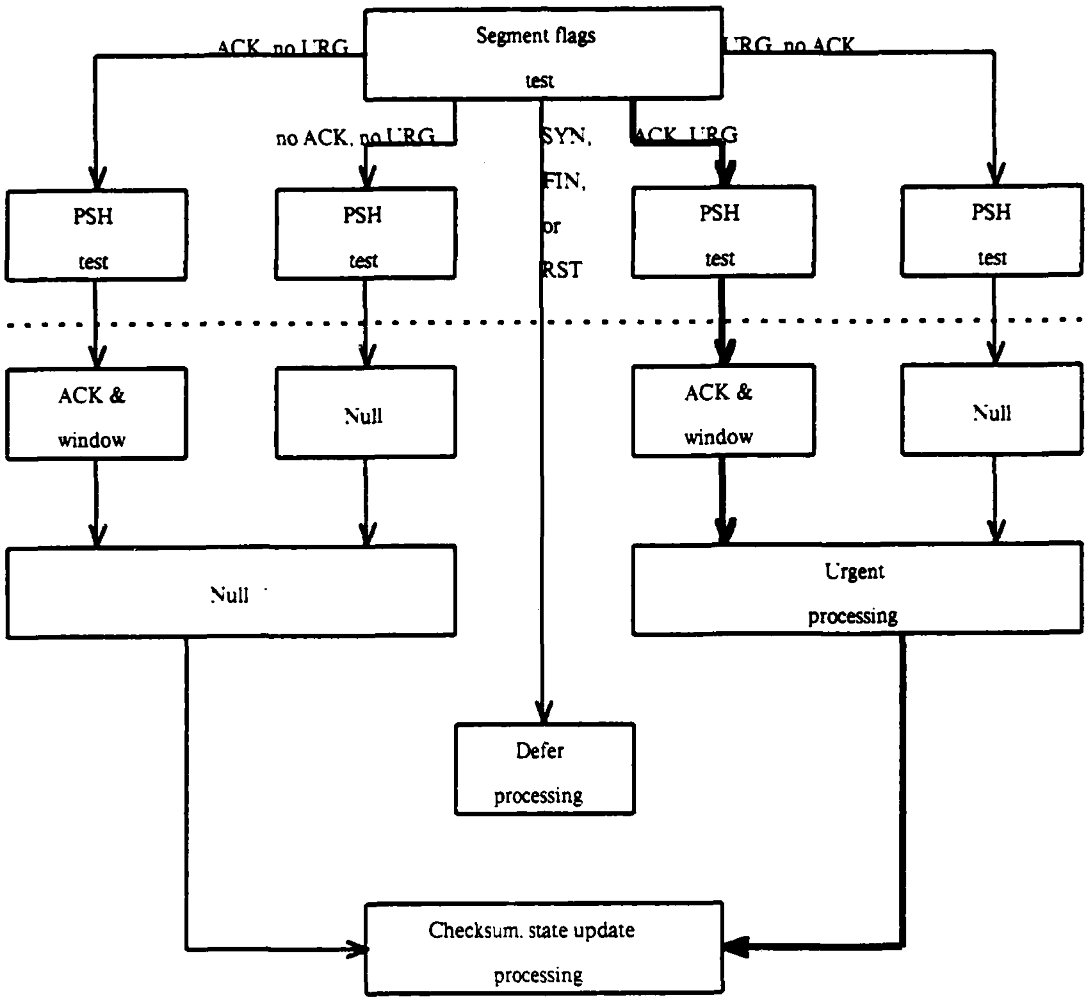


Figure 5-12: TCP filter code major states (part 2)

utilized near capacity. Overall, the filter program for the worst case segment requires slightly less than 50 percent of the available 96 ALU operations; however, the demand is highly variable. For example, there is little use for ALU operations during the first 8 bytes of the segment because the operands for computation are not available until the segment sequence is accumulated; once the sequence value is available, the ALU is heavily used. Hence, it would be impossible to even out the ALU load so that only one ALU operation per instruction would suffice.

Register usage

The PC and shift registers will probably have to be separate from the main register file because of their special properties. The register file must hold the connection record (11 words), the prompt acknowledgment, two checksum registers, the TCP segment length register, FR, CR, and temporaries for values such as `toxfer` and `skip`. Without inserting delays in the filter code to allow registers to be reused, 16 registers is not enough and hence a 32-register file and 5-bit register numbers are used.

Instruction formats

The light ALU load and the repetitive nature of certain filtering computations can be used to reduce the complexity of filter machine instruction decoding. One of the two ALU operations for a single instruction (termed ALU operation 1) can be restricted to a small number of choices. In the TCP filter code developed later, ALU operation 1 is restricted to be either null, the high-order continuation of a double precision operation specified by ALU operation 2, or a checksum addition.

Using this rule, an instruction required approximately 18 bits:

- Two bits for ALU operation 1.
- Thirteen bits for ALU operation 2, consisting of a 3-bit opcode, and 5 bits each for the 2 operands, where the destination is the same as the first operand.
- Three bits to specify the memory operation (This necessitates load and store operations into a subset of the filter machine register file and subsequent copying to the desired register via an ALU operation 2).

Optimization of instruction encoding is beyond the scope of this report; denser encodings are clearly possible through the use of two-level microstore, etc. This encoding does demonstrate that instruction information doesn't require a horizontal format.

Worst-case filter code sequence

Figure 5-13 shows the filter code sequence for the worst-case TCP segment. The worst case segment includes new acknowledgment, window, and urgent values which replace old values after the maximum number of tests. The worst case segment also contains two bytes of data so that truncation due to receive window size is possible. The results of conditional branches are shown in parentheses in the memory-operation column.

This code-execution history is also the mainline of the TCP filter program. It frees memory operations which would otherwise be used for branch instructions and FR reloads. Other code sequences will have fewer computations to perform and hence more free memory cycles for branches to merge code sequences. In particular, the memory operations following the data-offset field will be consumed in reloading the FR.

The first eight instructions locate the connection record for the arriving segment. The last instruction loads CR (SIN) with the address of the connection record. No other processing is possible because the connection record is not located until the end of the first eight instructions.

During the eight instructions corresponding to the segment's sequence value, the connection record values are read into the filter machine register file. The filter machine accumulates the incoming sequence field in registers `seqh` and `seql`.

The majority of sequence-value processing is performed during the arrival of the segment's acknowledgment-sequence value. Computation equivalent to the sequence and data pseudocode is performed during these eight instructions and the instruction corresponding to the data offset field.

After these instructions, code sequences are merged during the reserved field. A memory operation is available here for this purpose. The value in `toxfer` is used to update `left`, `bufad`, and `window`. Note that a value of zero preserves the old values.

		ALU OP 1	ALU OP 2	Memory operation	N reg
0	Source			CID N-way	CR
					CR
1	Port				CR
		C2=C2+SR			CR
2	Destination				CR
					CR
3	Port				CR
		C2=C2+SR	C1=0		CR
4				input sout	
	Sequence			input leftl	
5				input lefth	
		C2=C2+SR	seqh=SR	input window	
6		skip=left		input bufad	
	Number			input rackl	
7				input rackh	
		C2=C2+SR	seql=SR	input rwindow	
8		skip=skip-seq		input urgpl	
	Acknowledge		toxfer=tcpdl-skipl	(>) packet overlap?	FR
9			toxfer:window	(Y) data left?	FR
		C2=C2+SR	seqackh=SR	(>) clip test?	FR
10			window		
	Number		ack=ack+CHOKE	(Y) window closed?	FR
11			toxfer=window	input urgph	
		C2=C2+SR	seqackl=SR	input state	
12	Data Offset		state=state+NEWCHOKE	(=) =5?	FR
	Reserved	left=left+toxfer			
13			window=window-toxfer	flag test 1?	FR
	Flags	C2=C2+SR	bufad=bufad+toxfer	flag test 2?	FR
14	Window		state=state+NEWPUSH		
		ackseq:rack		output leftl	
15				(=) newer ack?	
		C2=C2+SR	wdelta=SR	output lefth	
16			wdelta:rwindow	input CIDptr	
	Checksum			(Y) bigger window?	FR
17			state=state+NEWWIND	output window	
		C2=C2+SR		output bufad	
18	Urgent			output rackl	
				output rackh	
19	Pointer		state AND UPEND	output wdelta	
		C2=C2+SR	udelta=SR	(Y) old urgent?	FR
20		seq=seq-udelta			
		seq:urgp		(Y) new urgent?	FR
21			state=state+NEWURGENT	output seql	
	Data	C2=C2+SR	tcpdl=tcpdl-2	output seqh	
22			C1-C2	(Y) data end?	
				(Y) checksums O.K.?	FR
23			sout=temp	out state	
		C2=C2+SR		output newptr	

Figure 5-13: TCP worst-case filter code sequence

As the segment's flags arrive, code execution has been consolidated into two main code sequences corresponding to the PSH sensitive and PSH insensitive cases. (Of course, many degenerate cases exist. For example, an unknown connection, strange sequence number, bit-timing error, or premature transmission completion uses other filter code sequences.)

The N-way branches associated with the segment's flags break each of the two PSH cases into 9 subcases each, for a total of 18 possible code sequences. One of the 9 subcases corresponds to flag fields which contain SYN, FIN, or RST bits. The other 8 cases correspond to the 8 combinations of PSH, ACK, and URGENT when SYN, FIN, and RST are all zero. ALU cycles during this period update the values of `left`, `bufad`, and `window`.

The PSH sensitive and insensitive execution sequences are merged by branch memory operations during the first four bits of the segment's window field.

Based on the arriving values, the instructions from the 14th through the 21st byte calculate the new values for the urgent, acknowledgment, and reverse window values and output them to the connection record.

The pointer swap that accomplishes the state update is performed after the checksum for IP and TCP is verified. The worst case for the data length is two bytes, the minimum length that can cause an overflow of the receive window. Assuming the checksum is correct, the pointer swap finishes the state update.

Storage requirements

The aggregate storage requirement for TCP filtering is the sum of

1. N-way branch tables for connection recognition. In the worst case, 8 tables of 16 words each are required per connection.
2. Connection state blocks. Using the same 16-word blocks as N-way branch tables, 2 blocks per connection.
3. Code and filter tables for sequence and data octet, acknowledgment, window, urgent, checksum and state-update processing. Approximately 2K words are required for this storage.
4. Miscellaneous code and filter tables for segments not processable by the filter, including segments with bad checksum, unknown connections, etc. Depending on error-recovery procedures, this cost is estimated to be 1-1.5K words.

The TCP static requirement is less than 4K words, and connections cost 160 words each in the worst case, including IP incremental costs. Thus 8K of memory is a reasonable minimum and will support at least 20 open TCP connections.

Alternatives and Extensions

The two main opportunities for enhancing the TCP filter code are enlarging the set of segments which the filter can process and improving the buffering system for data octets. The practicality and benefits of these enhancements will depend on the host's needs, since both changes depend on the usage patterns of the host's processes.

Enlarging the set of segments processed by the filter

The TCP filter code described in the previous sections handles all segments except for those which contain RST, SYN, FIN, or options, and those segments which cannot be bound to a connection state block. Since the only defined TCP option, other than a NOP and an "end of list" option, is restricted to SYN-bearing (i.e., initial connect) packets, and since packets which cannot be bound to a state block typically occur during initial connect, enhancements will only improve performance during the opening and closing of a connection. The filter can't conveniently maintain the filter tables used for connection recognition because of the real-time constraints on the filter's operation and the limited instruction repertoire of the filter; hence any improvement is further restricted to those packets which don't need to construct filter tables for connection recognition.

The sequence of packets involved in the three-way handshake for initial connect is typically composed of a SYN packet, a packet containing an ack for the first SYN and a reverse SYN, and an ack packet for the second SYN. Since both of the SYN packets involve connection setup, they probably will need interface microprocessor attention in any case; hence there is little reason to even attempt to process them competely in the filter. The third packet of the three-way handshake doesn't need to carry a SYN, and hence can already be processed by the filter. Thus there is no justification for additional filter support during connection opening unless the TCP implementations retransmit already acknowledged SYNs with later data octets.

Accepted RST-bearing segments cause user notification and connection closing and hence need interface microprocessor attention. The remaining RST segments are sufficiently rare that filter support is also not justified.

FIN-bearing segments are a more practical expansion, since the connection deletion is caused by a timeout and not by the FINs themselves, and since a half-closed connection may well receive an indefinite number of FIN-bearing segments which need no special processing. A reasonable extension would be to defer processing of a FIN packet only if it is the first FIN on that side of the connection, and otherwise to ignore the presence of the FIN. A bit in the connection record state variable could be used to signal that FINs are to be ignored; alternatively FINs could be ignored if they don't update the receive left edge.

Improving the buffering system for data octets

A more fruitful area for improvements is the buffering mechanisms used for received data.

At a minimum, the the TCP filter code should be changed to use a circular buffer for its internal buffering of data octets. This is a small change to the filter code since it only requires a test to divide `toxfer` into two sections when the buffer address wraps around its high limit. (This procedure avoids the need for wrap around tests after each data octet transferred into the buffer area.)

For filter interfaces with direct access to user buffers, the filter code could be modified to transfer data directly into the user buffers, bypassing the circular buffer in the interface as long as a user buffer was available. This change is more computationally complex, and would probably add to the filter's processing time, although the additional time could be easily masked by the interpacket gap of a bus system.

5.4 LOCAL

The Design Goals for LOCAL

The LOCAL protocol discussion is not intended to be an argument for some new protocol; instead LOCAL is a vehicle for showing that the filter can support message protocols which are less connection oriented than IP/TCP ("message LOCAL") and also for showing that the filter can support different encapsulations of high-level protocols (TCP is used as an example, hence "TCP LOCAL"). Although LOCAL uses the same packet format for both of these functions, the two functions are almost completely independent; hence the filter code to support the two functions is also almost completely independent.

This section emphasizes applicable techniques rather than specific implementations, as LOCAL would presumably be tailored to fit a particular environment. In the following discussion, low-level issues dealing with the design of filter code, acknowledgments, etc., are omitted where they follow the principles previously discussed for IP or TCP.

Message LOCAL is patterned after the DCS message-transfer facility described in chapter 3. Message LOCAL maintains a fixed-size state block for each process it serves. Within the process-state block, two groups of connection-state records (sending and receiving) hold the connection state for individual process-to-process connections. In the normal case, the LOCAL filter code allocates and deallocates connection records using a "working set" approach similar to that used in the DCS system; LOCAL is less connection oriented in that the host and interface microprocessor are less responsible for maintaining connection state. The key filter code issues are the algorithms which locate a connection record for arriving packets, the algorithms which manage the working set, the algorithms for processing user-initiated messages, and the algorithms for processing for purge and purge-request messages.

The multiple encapsulation goal is illustrated by showing that TCP packets can be encapsulated in LOCAL as well as IP. The key issue is the extent to which the TCP filter code discussed in the previous section must be aware of the encapsulating protocol.

Ack Development

Ack development in LOCAL has two parts, corresponding to message LOCAL and TCP LOCAL. Rather than define new prompt acknowledgments, both types of LOCAL use the prompt acknowledgment format defined for the IP and TCP. Message LOCAL generates a prompt acknowledgment which looks like a combination IP/TCP prompt acknowledgment; TCP LOCAL sets the IP prompt acknowledgment bits and lets TCP processing fill in the TCP prompt acknowledgment bits. In a sense, message LOCAL is both a datagram and a transport-level protocol, whereas TCP LOCAL is simply a datagram protocol. This policy simplifies TCP LOCAL, but is merely a matter of taste in the case of message LOCAL.

The LOCAL prompt acknowledgment bits corresponding to IP prompt acknowledgment bits are translated as follows:

1. *IP header checksum error => LOCAL header checksum error.* For LOCAL, this bit is always unset because the LOCAL header contains no checksum for header data. Although a header checksum could be added to LOCAL, there is little benefit in doing so since LOCAL will never be used outside of a single-hop environment. All message LOCAL packets are protected by the FRAME level checksum; the LOCAL addresses of TCP LOCAL packets are used to calculate the checksum of the TCP pseudoheader.

2. *Destination address recognized.* No change is required. For message LOCAL, this bit means that the filter could associate a process-state block with the arriving packet. For TCP LOCAL, this bit means that the connection search was able to successfully chain through N-way tables corresponding to the LOCAL address fields. The encoding of LOCAL addresses is discussed under the "triage" section for LOCAL.
3. *Protocol recognized => LOCAL filter running.* Message LOCAL always sets this bit; TCP LOCAL sets this bit when it recognizes a TCP encapsulation.

The LOCAL prompt acknowledgment bits corresponding to TCP prompt acknowledgment bits are translated as follows:

1. *TCP checksum error.* For TCP LOCAL, this bit signals a TCP checksum error and implies that all other TCP prompt acknowledgment bits are to be disregarded. For message LOCAL, this bit follows the FRAME-level checksum error.
2. *Connection not found.* For TCP LOCAL, no change is required. For message LOCAL, this bit means that a connection-state entry could not be found or created.
3. *TCP processing deferred.* No change is required. For message LOCAL, this bit now means "LOCAL processing deferred."
4. *Control value ACK (CACK).* For TCP LOCAL, no change is required. For message LOCAL purge and purge-request messages, it means that the specified action will be taken. For message LOCAL data messages, this bit follows the ACKALL bit.
5. *Acknowledge all data (ACKALL).* For TCP LOCAL, no change is required. For message LOCAL, this bit signifies that the message has been copied, i.e., it has the same semantics as the DCS **accept** bit.
6. *Receive window now closed (CHOKE).* For TCP LOCAL, no change is required. For message LOCAL, this bit signifies that the network interface will not accept additional data messages.

Triage

The top level of triage in LOCAL separates message LOCAL packets from TCP LOCAL packets; the separation is controlled by the 48-bit LOCAL destination address field. Every 48-bit LOCAL address starts with a 4-bit code which separates encapsulated packets (code = 0) from message LOCAL packets (code = 1). This leaves 14 choices available for future expansion. In addition to this code, TCP LOCAL destination addresses carry an 8-bit protocol field and a 32-bit IP address; message LOCAL addresses carry a process name and control bits. In order to keep addresses the same size, a 32-bit process name is assumed. The two formats are shown in Figure 5-14.

The 48-bit TCP LOCAL address contains

1. A 4-bit code which identifies the packet as a high-level encapsulation.
2. An 8-bit code which identifies the encapsulated protocol (e.g., TCP), using the same encoding as IP uses for the corresponding field.
3. Four bits of unused space.
4. A 32-bit IP address.

4	8	4	32
0	TCP		IP address

LOCAL address for TCP encapsulation

4	4	8	32
1	MB	MDF	Process name

Message LOCAL address

Figure 5-14: LOCAL address formats

The 48-bit message LOCAL address contains

1. A 4-bit code which identifies the packet as a LOCAL message.
2. A 4-bit field (labeled MB in Figure 5-14) which holds the start of message (SOM), end of message (EOM), and sequence bits.
3. An 8-bit field which holds the message definition field (MDF) field. This field is a code which identifies the message as being either a data message, a control message, a purge message, or a purge-request message.
4. A 32-bit process name.

Lower level triage for TCP LOCAL packets follows the conventions established for TCP; for message LOCAL, the following low-level triage is performed:

1. Packets with a bad checksum cause a prompt ack to be sent with the IP checksum error bit set. The packet is discarded.
2. Packets which are addressed to an unknown process are ignored; no prompt acknowledgment is generated.
3. Data and control messages which have a new sequence value and which can be copied are fully acknowledged and queued for the host process. The filter selects a new packet buffer for the next packet on the medium.
4. Data and control messages which have old sequence values or are received when no buffers are available are acknowledged. The packet is discarded.
5. Purge and purge-request messages cause modification of the process state block. Because they can be processed immediately, they are always acknowledged and discarded.

Activities and Code

Figure 5-15 shows the data structures which are added to the filter data base to support message LOCAL.

The process-state block contains all of the communication-state information for a particular process. This data includes information common to all connections, as well as separate connection-state blocks for every active connection. The common data includes pointers to the queues for data

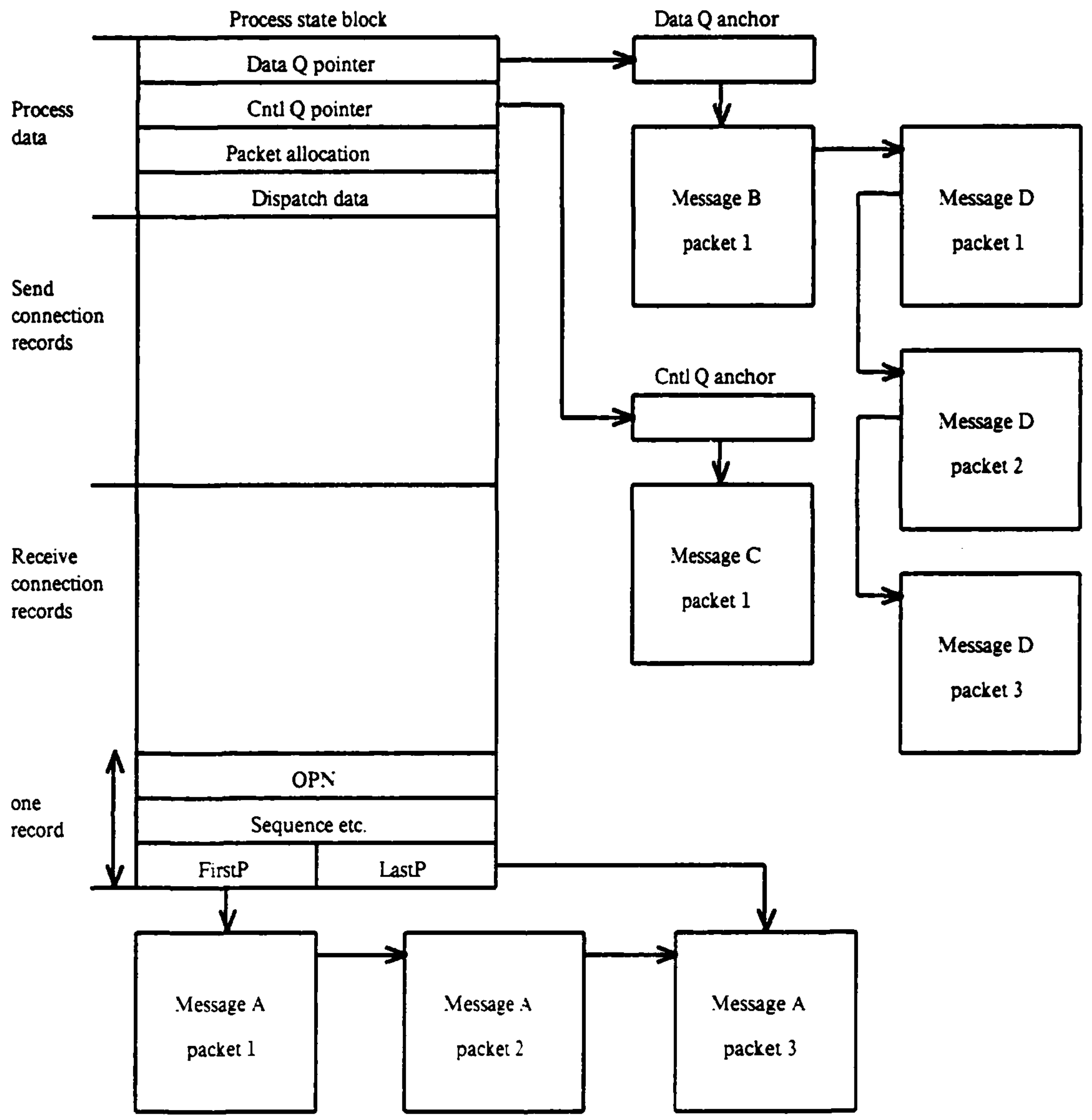


Figure 5-15: Message LOCAL data structures

and control messages, rather than the queue pointers themselves, so that messages for different processes can be combined. This is typically useful only for connection messages, where a single system process may supervise several inferiors. Messages on these queues are chained in a FIFO manner; multipacket messages have internal pointers to link all of their packets. Multipacket messages which are not yet complete are queued on the appropriate connection record.

LOCAL connection search

Local processing begins by branching to 1 of 16 connection-search routines based on the first 4-bit field in the destination address. This implementation defines two routines: encapsulation LOCAL (TCP is not identified until the protocol field is input) and message LOCAL. The other 14 values defer processing to the interface microprocessor which handles the erroneous packets.

The encapsulation search checks for a known protocol field (i.e., TCP) and then uses the normal IP/TCP connection-search tables to locate the TCP level connection block. If the destination-address search fails, the packet is ignored; if the origin-address search fails, the packet is deferred. As this search takes place, the TCP pseudoheader checksum is computed based on the appropriate sections of the LOCAL address fields.

The search routine for message LOCAL begins by locating the process state block corresponding to the LOCAL destination address. This search uses the same N-way table approach used for IP/TCP connection recognition. If the destination-address search fails, the packet is ignored.

Once the process-state block is located, the message LOCAL search routine must locate the connection record corresponding to the origin address. Although almost any search technique could be used, a linear search is adequate if the process-state block holds only a few connections. For a 32-bit wide search, each comparison requires two filter memory cycles to load the candidate entry's name, two filter ALU cycles to perform the comparison, and one filter memory operation for the conditional branch; since each filter instruction specifies two ALU operations for every memory operation, the three filter memory cycles per comparison are the limiting factor.

During the accumulation of the origin-process name, the filter needs no memory cycles; it can use these 12 cycles to preload comparison values. Thus each of the 4 memory cycles during the LOCAL length field can be used for conditional branches. Thus the first 4 process names don't add any delay; by left adjusting the 32-bit origin process name in the local origin address field, 5 comparisons can be done without inducing any delay. In the DCS, the average process uses 4 connections: 1 connection each for the controlling process, input, the terminal, and output connection. Hence, 5 active connections per process should not be a problem. Expanding the size of the table beyond this limit induces 12 bits of delay per name.

<u>Message LOCAL working set management</u>

A small fraction of processes will require more connections than can be recorded in the process-state block. These processes cause the interface to open and close active connections, although these activities are not visible to the process. To make this scheme efficient, the interface needs a replacement strategy which purges the least active connection from the process-state block when a new connection record is needed.

The first replacement criterion is that the connection record is not locked and does not have incomplete messages queued. This criterion avoids buffering and deadlock problems associated with incomplete message packets and also avoids ambiguities which can be caused by a restart of the distant process.

The second criterion is that the filter should select the least recently used (LRU) connection of the set that pass the first test. This implies that the interface must keep track of connection usage; ideally the connection records should be totally ordered with respect to their most recent usage. Although this policy could be implemented, a simpler strategy is to include a usage variable in each connection record and increment the variable each time the connection is used. In order to avoid preserving connections with high, but not recent, usage all of a process state block's usage variables are shifted right by a constant factor each time any connection record for the connection is purged. Thus unused connections will have their usage go to zero as other connections are purged. A new connection can be setup with a high usage value to avoid immediate purge. By comparing the usage values, the least used connection record can be selected; filter processing time to maintain usage statistics is limited to the two filter instructions needed to increment the usage variable.

Message LOCAL purge and purge-request messages

Purge messages are bound to connection records by the search discipline already discussed. A purge addressed to a nonexistent connection record for an existing process is acknowledged to accommodate process restarts. Purge messages addressed to a connection that is locked or has an incomplete message queued are deferred. Normally, the interface microprocessor will resolve this condition by closing the connection with an error to the process. Purge messages to other existing connections cause the connection record to be discarded.

Purge-request messages to locked connection records set a bit in the connection record which is noticed by the interface microprocessor when the multipacket transmission in progress completes. All other purge requests are deferred.

Data and control message processing

After binding data and control packets to a process-state block, the filter performs several other tests which set prompt ack bits and determine whether the packet is discarded, deferred, or queued for host processing. Although the actual filter code for this processing intermixes the tests to optimize processing time, the actions of the filter are best explained in terms of sequential steps.

The first step tests the sequence in the packet's MB field against the sequence value in the connection record. If the packet's sequence is old, the packet is discarded and a prompt ack is returned with ACKALL set. If the packet's sequence is new, the packet is passed on to the next step; note that the connection record's sequence value is not updated until later (if at all).

The second step is to verify that a buffer allocation exists for this packet. (The lack of such a mechanism was one of the DCS's worst problems.) In this implementation, buffer allocations are on a per process basis; the filter code need only check the allocation field of the process-state block. Other implementations might choose to maintain separate data and control allocations, allocate buffers on a per connection basis, or some other scheme. If an allocation is available, the prompt acknowledgment is set to include ACKALL and the sequence bit in the connection record is incremented; otherwise the packet is discarded.

The next step performs any required reassembly of packets into messages. Normally the packet will be a complete message in itself (SOM and EOM set) and will require no further action other than to verify that a partial message is not already queued. If the packet is a fragment that completes a message, it is united with its predecessors and the chain is passed on. If the packet is a fragment which doesn't complete a message, it is queued off of the connection record. Errors, such as a packet with SOM arriving on a connection which already has fragments queued, cause the packet to be deferred.

The last step of processing takes the complete message and queues it on either the data or control message queue. The queue pointers are available in the process-state block. After the message is queued, implementation-dependent processing uses the process-state dispatching parameters to decide whether to cause an interface microprocessor or host intervention request.

TCP LOCAL encapsulation

Given that the message LOCAL filter code locates the TCP connection block and calculates the TCP pseudoheader checksum, the TCP filter code need only be given the value corresponding to the TCP segment length (`tcpdlen`), to complete a transparent use of LOCAL encapsulation. This value can be easily calculated using the LOCAL length field.

Alternatives and Extensions

The main shortcomings of the LOCAL protocol as described are its inability to handle broadcast process names and its reliance on the interface microprocessor for purge and purge-request service.

The source of the broadcast-addressing problem is that broadcast messages need a separate connection record for each address; if process A receives both uniquely addressed and broadcast messages from process B, A must maintain separate connection records for the two types of transactions. The difficulty is not allocating the separate records, but the doubling of the width of the comparisons which search for the connection record in the process state block. Doubling the length of this comparison either doubles the search time or requires that only half as many connection records are used in each process-state block.

One approach to the broadcast problem is to use shorter process names. A second approach is to improve the purge and purge-request filter code to lower the cost of opening and closing connections; however this approach can be ineffective since closing a broadcast connection can require many retransmissions to guarantee that all destinations receive the purge. The only satisfactory solution seems to be to create a separate process-state block for broadcast names and remove the ambiguity as to the destination address of the packet; this approach solves the timing problem but is potentially expensive in terms of interface memory.

Increasing the scope of filter processing for purge and purge-request messages is quite reasonable, given that the format of these messages can be redefined to suit the implementor. In order to increase the amount of filter processing available for these messages, the purge and purge-request messages can be padded to artificially increase the processing time available.

5.5 BROADCAST AND MULTICAST BINDING

Historical Problems with Broadcast and Multidestination

Interprocess communication need not be restricted to a single destination; information from a particular source often needs to be distributed to several destinations. Two forms of this type of communication are broadcast, in which a given transmission goes to all destinations, and multicast or multiple destination, in which information is sent to a specified subset of all destinations. In practice, pure broadcast has few applications; broadcast algorithms are used in situations where addressing, the communications channel, or some other mechanism acts to restrict the distribution of data. The most common implementation of multicast consists of a broadcast transmission to all hosts in the network followed by individual selection mechanisms in each host.

Whether implemented implicitly or explicitly, multicast transmissions are inherent in any application which includes a distributed database [Gifford 79a, Gifford 79b, Rothnie 77, Stonebreaker 79, Thomas 79]. Intuitively, multicast transmissions search for an object among a set of possible locations or inform all concerned parties of an event. Multicast queries enable multiple database servers to process queries in parallel; some multicast addressing systems permit the requester to be ignorant of the distribution of data among a set of database servers. Multicast transmissions are useful in the update process; multicast allows for rapid update of redundant copies as well as rapid distribution of "ballots" in voting systems.

In almost all contemporary networks, neither multicast or broadcast services are available to the user as part of standard communications protocols. Processes must simulate multicast via sequential transmissions to all members of a broadcast set or by various algorithms that pass messages around a software ring of processes.

Those local networks which do support multicast use specially encoded addresses for packets destined for a multicast set. The encoding defines special values for particular address field components (usually the host field, rarely the network number as well); the remaining address field components (socket, etc.) must be well-known values. One special value means "all" and can be used to send a packet to the same socket number on all hosts, creating a broadcast capability. Other special values, frequently called "logical" host addresses, are defined for each multicast set. Packets addressed to a logical host are recognized by all physical hosts which have processes in the multicast set. Each host accepts packets addressed to its physical host address, the broadcast or "all" value, and all logical addresses with resident multicast set members.

Multicast features are not supported as standard parts of contemporary communication protocols for several reasons:

1. The architecture of long-haul packet-switched systems makes multicast very expensive to implement; restricted bandwidth, complex topologies, and limited processing power in switching nodes all contribute to this problem. These constraints also make it difficult to design an algorithm for distribution which works reasonably for both large and small multicast sets. Algorithms for both cases have been developed and analyzed [McQuillan 78, Dalal 77, Dalal 78] and tend to impose sizable computational and bandwidth costs, especially when reliable transmission is required. Hence, there is no existing base of experience with multicast that can supply any sort of intuition regarding the uses and design of such a facility.
2. Although there are several systems that could profit from a multicast facility (e.g., RSEXEC, network mail, network speech), each individual application doesn't justify the cost of integrating multicast into the existing protocol structure. Applications which use this type of communication, such as routing updates [McQuillan 80], mail, and name servers, use custom programming support.
3. Multicast systems require new code to allocate multicast addresses, distribute the multicast addresses to members of multicast sets, and reclaim addresses when the set is no longer in use. This overhead limits multicast to situations in which the multicast channel will be in use for a long period and the group's population is static. In practice today, multicast sets are assigned well-known addresses though administrative, rather than dynamic allocation.
4. The potential performance benefits of multicast derive from the use of a single transmission to deliver a packet to multiple hosts. In practice, these benefits are limited

by the necessity to return a separate acknowledgment for each multicast recipient. Although most of the distributed database systems in existence [Gifford 79, Stonebreaker 79, Thomas 79] incorporate high-level error control to deal with failed hosts and communications paths, these recovery procedures are not sufficiently efficient to use for transient transmission failures.

The benefit of multicast approaches a 50 percent reduction in transmissions when compared to the cost of using multiple single-recipient transmissions. This benefit must be compared to the additional cost for the multicast channel state information, as the processes in the group will require simple single-recipient connections to return the results of multicast queries.

5. The protocol mechanisms in standard protocols used for sequencing, flow control, etc., are oriented toward one-on-one communication and will not work in a multicast environment without modification.
6. A fundamental problem with the simple encoded address scheme is that what is really wanted is a communication channel to a set of processes defined by some semantic constraint (e.g., "all file handlers," "all processes using the accounting database") rather than a list of process addresses. In the terminology of networks, we want to use "names," which identify what we seek, rather than "addresses," which identify where it is.

   Current efforts at providing name servers to solve this problem for single name-to-address mapping suggest that creating a universally consistent and useful name space is unrealistic. A somewhat less ambitious goal, which can be realized, is to allow a set of processes to construct their own name space as a subset of network addresses. Such a system can use encoding techniques to reduce the length of names to reasonable size for a particular domain. In addition to the component systems used in existing systems, several systems have expressed the need for boolean predicates encoded in the address [Pardo 79, Rowe 79].

The filter machine can provide substantial efficiency improvements which will promote the construction of reliable multicast addressing and transmission primitives.

Multicast Addressing and the Filter Machine

The filter machine can easily be programmed to recognize conventional multicast set addresses; these addresses can be entered into the filter data base in the same way as normal addresses. With the filter, the only limit on multicast sets is the amount of filter memory needed to store the filter data base, rather than the size of an associative memory, worries about reducing throughput due to broadcast messages that will be discarded, etc. Because the address-recognition mechanism in the filter is totally programmable, any of the existing schemes can be used.

The filter can also support more powerful multicast addressing schemes, such as those proposed in [Pardo 79], [Rowe 79], and others. The essential difference between these schemes and the conventional multicast set addresses is that the new schemes require a more complex address comparison rule than simple equality.

Rowe [Rowe 79] describes an addressing scheme for distributing queries to multiple database servers. The intent of this scheme is that query addresses should carry sufficient information to select only those database servers which contain relevant data. Queries use a custom address that has one component which identifies a particular database and a second component which describes the categories of information in the query.

The database ID has a standard format and could be a pseudohost or channel identifier compatible with other packet formats. In general, the category component can only be interpreted in the context of a particular database ID; because databases have different numbers of indices, the category component should have a large fixed size or a variable size defined by the application database.

For example, the database ID component might specify "the personnel database" or "the accounts receivable database." The second component of the address could be a bit vector in which each bit is associated with an index of the database identified by the first component. For example, if the first component specified the "personnel database" the second section might have bits for "name," "sex," "address," "phone number," etc.

The first component limits distribution to those database servers which handle a specific database; the second component limits distribution to those databases which have information for the specific query. Using the previous example, if the database ID component specified the "personnel database," and the second components had ones in the "name" and "sex" fields, the query should only be distributed to personnel databases which contain name and sex information, and not to other database servers, or to personnel database servers which do not have name or sex data.

When a network interface sees such a multicast packet, it checks the database ID against the list of databases in the attached host. If the ID is found, the interface uses the category component in the packet and a category descriptor associated with the host database to decide whether to accept the packet. Rowe's paper [Rowe 79] was based on the NS LNI, and partitioned the 32-bit LNI address into the two components. The component test ANDed the category bits in the packet with the category bits in the name table entry of the LNI.

With the increased capabilities of the filter, a much more powerful multicast query distribution system could be built.

- The address size of the category component can be variable. This could allow different category sizes for a particular database or even different category sizes for the same database.
- The filter can be programmed to increment the sequence space associated with a multicast packet which has a new sequence, but fails the category test. Using this policy, separate sequence spaces or host sequencing can be avoided. A similar scheme could allow multiple query sources to share a channel without requiring separate sequence spaces for each source.
- The category comparison rule can be extended to include any finite state language, as well as some arithmetic, and still be performed as the packet arrives. The packet could even carry the rule as a subfield of the category component. In the previous example, messages could be addressed to all databases which have name and sex information, name or sex information, salaries greater than $50K, etc.

Reliable Multicast Transmissions and the Filter Machine

Many different algorithms can implement reliable multicast in a local network environment. The best choice depends on many factors, including the type and reliability of the medium, the capabilities of the network interfaces, the number of hosts on the network and in the multicast set, and the relative costs of medium bandwidth, host processing, and time to complete the multicast transmission. Some algorithms are usable only in a restricted set of circumstances; for example, a ring medium might be required.

This section compares the performance of several multicast algorithms. The two metrics for comparison are the total number of transmissions to complete the multicast, and the total number of packets which must be processed by all hosts. (Packets discarded by the filter don't count.) The comparison counts a transmission and its prompt acknowledgment as a single transmission.

Four types of multicast algorithms are considered:

1. Simulation algorithms, which achieve the effect of multicast using separate one-to-one transmissions.
2. Multiple acknowledgment algorithms, which distribute the multicast text using some sort of one-to-many transmission, but collect acknowledgments via one-to-one transmissions.
3. Saturation algorithms, which rely on statistical arguments to avoid the need for acknowledgments following one-to-many distribution transmissions.
4. Negative acknowledgment (NACK) algorithms, which use filtering and negative acknowledgments.

Three levels of interface capability are used for comparison:

1. *Broadcast.* Transmissions can be addressed to a single destination or to all destinations.
2. *Multicast.* Transmissions can be addressed to a single destination or to a multicast address which is recognized by all interfaces in the multicast set.
3. *Filter.* All interfaces include a filter machine which can discard packets without host intervention. Except for the NACK protocol, filters generate prompt acks only for one-to-one transmissions.

Multicast by simulation

The most straightforward way for a network which lacks any sort of one-to-many transmission capability to simulate multicast is for the sender to transmit separate messages to each destination and receive separate acknowledgment transmissions in return. Each destination requires two transmissions, so that a multicast set of N destinations requires 2N transmissions. Each transmission is created and received by a host, so a grand total of 4N packets are processed by all hosts. This method is usually the most expensive, but can be used with any medium or system of protocols. It requires all senders to maintain lists of multicast set members.

Adding a filter machine to all hosts replaces host-generated acknowledgments with prompt acknowledgments, and hence saves 50 percent of both metrics.

An alternate method is to use a software ring of destinations: the source transmits the message to the first destination, which forwards the message to the next destination, etc. The last destination returns the message or an acknowledgment to the sender. If intermediate acknowledgments are not returned, a total of $N + 1$ transmissions are required. The advantages of this scheme are the reduction in traffic, and the fact that all of the destinations in the ring can transmit to the group without maintaining a list of all members; each member need only remember the next member. The drawbacks of this scheme are its slowness, since all transmissions are made in series, and its unfavorable performance in the presense of transient errors or host failures. No performance advantage is added by using a filter in interfaces using this scheme.

Multicast by separate acknowledgment

The archetypes of multicast algorithms for a local network rely on various types of one-to-many distribution followed by one-to-one acknowledgments. The message is distributed in one transmission; N acknowledgments are subsequently returned. The host packet cost includes one packet to originate the message plus 2N packets for the acknowledgments, unless the filter processes the acknowledgments, in which case only N acknowledgment packets are processed by the host. In general, the cost of acknowledgments is greater than the cost of the message distribution.

The main variability in cost is due to the different number of hosts which may receive the one-to-many message. In a network with C hosts, a message addressed to the broadcast address is seen by all C hosts; if a multicast address is available, only the N destinations see the message.

These algorithms can be used with either a bus or a ring. However, the controlled access provided by a token system avoids the likely collisions between acknowledgments for a bus system. The collisions are inevitable unless special pains are taken to spread out the acknowledgments; in the simplest CSMA system, these acknowledgments will always be precisely synchronized by the distribution transmission. The only remedy is to trade delay for collision avoidance.

Multicast by saturation

Saturation algorithms do not use acknowledgments; instead they make a statistical argument about the probability of transmitting at least one copy of the message to every member of the broadcast set. The basic principle is to transmit enough copies of the message to insure that at least one copy gets through to the destination.

If E is the probability of failure for one or more destinations (i.e., the desired overall error rate of multicast transmissions), and F is the probability that a transmission by one interface can be received correctly by some other interface, then we wish to solve for M, the number of transmissions which are required to send to a set of N hosts with probability E of failure. The probability of success for each host is

$$1-F^M$$

hence for all N hosts

$$1-E = (1-F^M)^N$$

solving for M

$$M = \frac{\log(1-(1-E)^{1/N})}{\log(F)}$$

Figure 5-16 graphs M rounded to the next higher integer for several combinations of E and F.

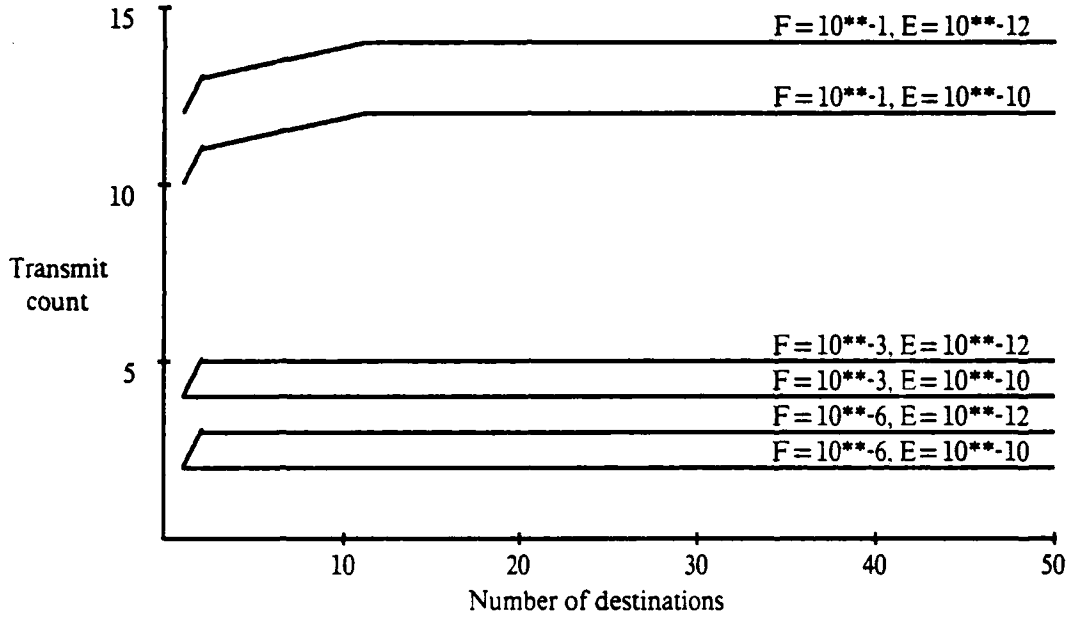


Figure 5-16: Multicast saturation transmission counts

Contemporary network designs typically strive for an error rate E of 10**-10 to 10**-12, although few empirical measurements have been made. Hence choosing a value for E is not a problem. The problem is determining the correct value for F, the interface-to-interface probability of packet loss. F is affected by several components:

1. Errors due to line noise, clock-recovery errors, transmitter-receiver errors, and other noise-related problems in the hardware.
2. Transmissions which are discarded due to half-duplex interfaces.
3. Transmissions which are discarded due to interfaces which have a "recovery" time following receipt of a message before another message can be received.
4. Transmissions which are discarded due to lack of buffer space in the interface or host.

The noise component is fairly well understood. Shoch reports that measurements on the Ethernet showed that a poorly designed interface has an error rate of 10**-3, whereas a well-defined interface had an error rate of less than 10**-6 [Shoch 79]. The point-to-point communication links used in a ring do even better.

The remaining components are difficult to quantify because they are dependent on host and network load, and the retransmission policy followed by the sender. The preceding analysis depends on the fact that the transmissions are independent trials with a constant probability of success; even if this is a good assumption most of the time in real systems, it is almost impossible to know F exactly. Filter interfaces have a real advantage since they can eliminate these causes of packet loss. Conventional networks can still use saturation algorithms, but only at an artificially high value of F.

For N = 1, M is equal to log(F)/log(E), that is, the ratio of the exponents of the error rates. M grows very slowly as N increases. Intuitively, this is because each additional destination benefits from the transmissions required by the other destinations.

The cost in terms of host packets processed is very sensitive to the interface capabilities. A saturation algorithm used with a broadcast interface results in every host on the network having to process M packets; similarly a multicast address has every member of the multicast set processing M packets. In both cases, M-1 of these packets are of no value and are discarded. A network of filtering interfaces is dramatically better because the duplicates are discarded by the interface.

These algorithms should work well for larger multicast sets when used with filtering interfaces. In other cases, the saturation technique is less valuable, although these algorithms are still attractive in situations where the multicast set is large; in situations where collection of acknowledgments can be difficult, such as an Ethernet; in situations where quick delivery is more important than host processing costs; and in situations where F can be estimated or where its precise value is not important.

Multicast by negative acknowledgment (NACK)

The strength of the saturation algorithm is that it avoids the need for N acknowledgments; its primary weakness is its inability to deal with interfaces which are unable to receive due to buffering problems and other load-induced conditions which may persist for indeterminate time. The independent trails assumption at the heart of the theoretical model is impossible to realize in practice.

This problem is avoided by negative acknowledgment (NACK) algorithms. Members of the multicast set which receive the message correctly don't transmit a prompt acknowledgment; member of the multicast set which would like to be able to copy the message, but cannot, send a NACK prompt acknowledgment. The NACK demands a retransmission by the sender of the distribution message. Because NACK generation requires no resources, interfaces can always generate NACKs. Thus load-induced conditions are removed from estimates of F, and F is driven by the noise-related causes which presumably are random.

NACK algorithms relate to separate acknowledgment algorithms in that the termination conditions are equivalent under DeMorgan's law. A separate acknowledgment system terminates on the basis of an AND condition over the individual acks; the NACK algorithm relies on the NOR of NACK transmissions. The functional difference is that the NOR predicate can be deduced from any single destination which sends a NACK; the AND requires all results. In use this means that the NACK protocol doesn't need to know how many destinations are in the multicast set; this is somewhat of a disadvantage in that it cannot detect destination failures.

In a bus system, multiple destinations may wish to send a NACK, and these transmissions will collide. However, the presence or absence of a transmission is all the information which is required.

In order to guarantee an acceptable level of reliability, the sender should retransmit until log(E)/log(F) transmissions have not resulted in NACKs.

In a ring network, the NACK system can be particularly reliable and effective. The sender transmits the distribution message followed by a "blank" message. If both return unchanged to the sender, then the multicast is complete. If any destination is unable to copy the distribution message, it overlays the blank message with a NACK. This scheme is superior to the **match/accept** bit scheme used in the RI and LNI in that it does not require the interface to examine bits of media data before it changes them. Instead, the destinations simply emit a constant message and its associated CRC.

The logical extension to this ring scheme is to encode whether the message is a multicast or not as a bit in the message; the interface hardware could invert the sense of prompt acknowledgments automatically.

Comparison

Table 5-1 summarizes the performance of these algorithms. Although the filter interface improves the performance of simulation and separate acknowledgment algorithms by a substantial amount, even more sizable savings are realized when saturation and negative acknowledgments are used. The ring NACK algorithm is optimal in that no fewer transmissions or host packet count is possible. The optimal transmit count is preserved in the Ethernet scheme, and only a small constant penalty is added to the Ethernet NACK scheme (on the order of 3 or 4 transmissions for existing Ethernet systems).

Figures 5-17 and 5-18 graph the transmit counts and host packets for various values of N. These figures show that the filter interface can provide substantial benefits for fairly small N. In situations where multicast speed and performance are important, the filter can be very valuable.

Table 5-1: Multicast algorithm comparison

Algorithm	Transmit count	Host packet count
Simulate	2N	4N
Simulate+filter	N	2N
Simulate(software ring)	N+1	2(N+1)
Broadcast+ACK	N+1	C+2N+1
Multicast+ACK	N+1	3N+1
Filter+ACK	N+1	2N+1
Broadcast Saturate	M	M(C+1)
Multicast Saturate	M	M(N+1)
Filter Saturate	M	M+N
NACK (ring)	1	N+1
NACK (Ethernet)	log E/log F	N+1

Legend:

C=hosts on network	M=saturation count
E=overall error rate	F=medium error rate
N=destinations in multicast set	

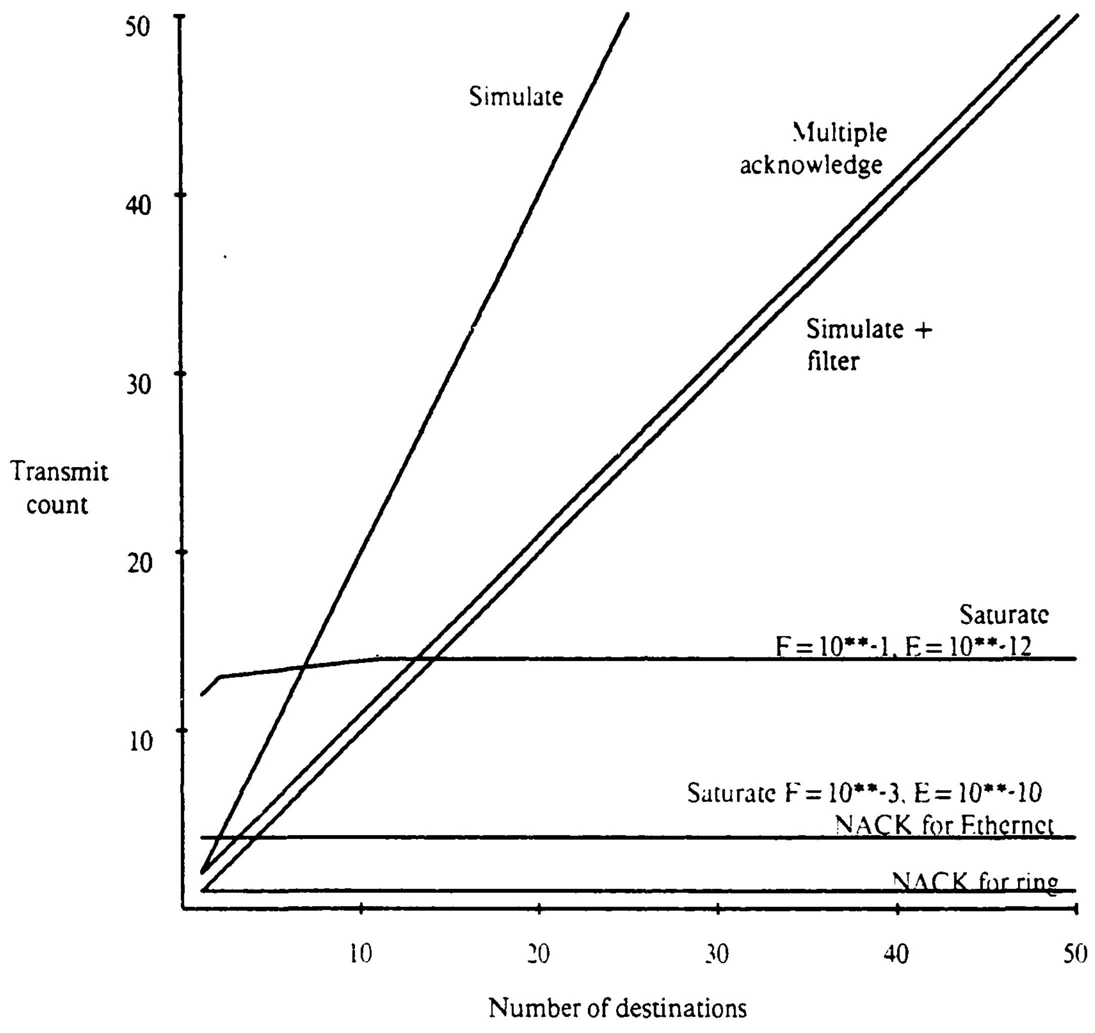


Figure 5-17: Transmission counts for multicast algorithms

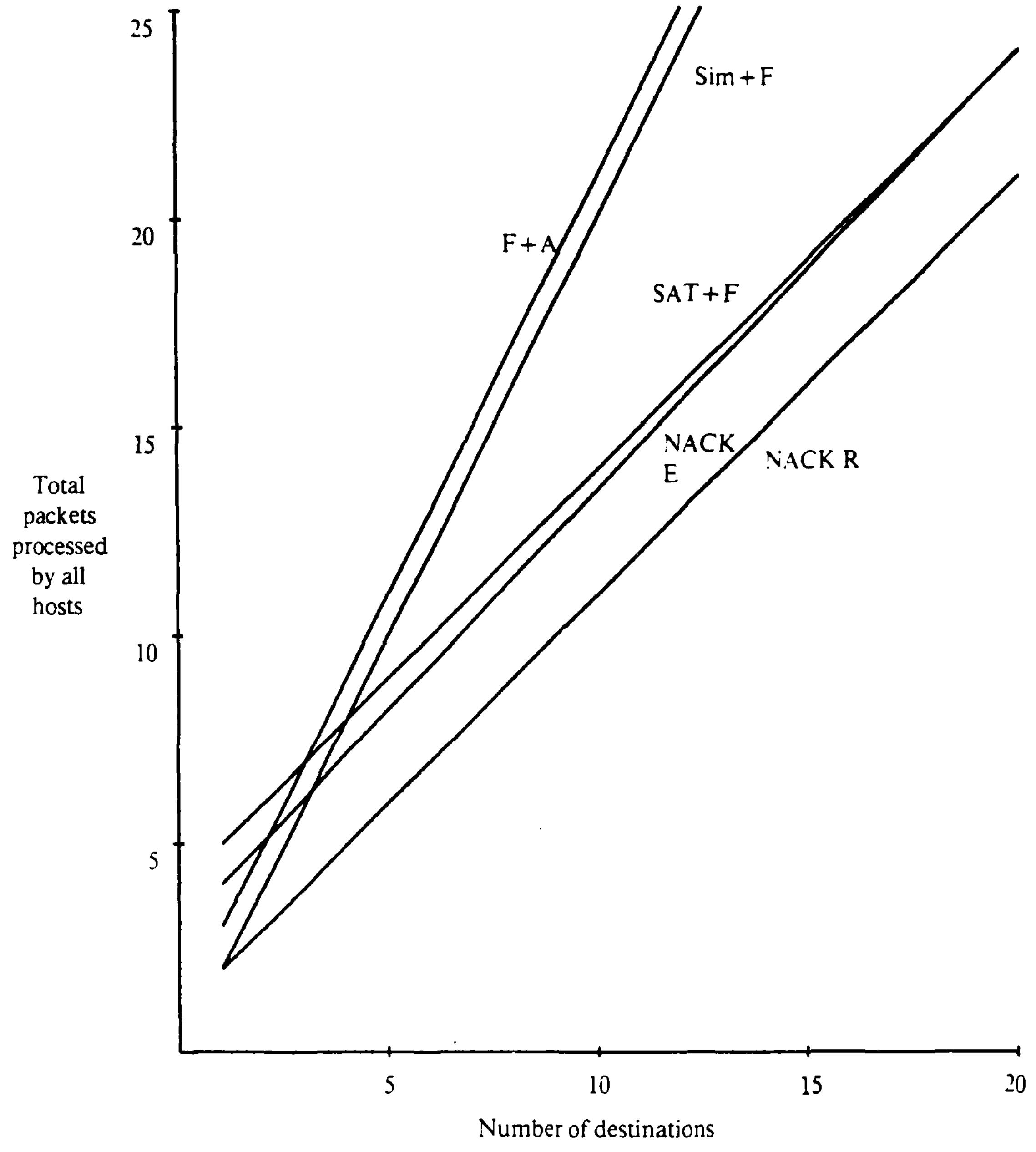


Figure 5-18: Host processed packet counts for multicast algorithms

6. DATA DELIVERY

6.1 INTRODUCTION

Data delivery, as defined in Chapter 4, includes all functions of the network interface which are not performed by the filter. The main distinction is that filter activities are synchronous with respect to data transfers on the medium, whereas data delivery activities need not be. The data delivery parts of the network interface coordinate the transfer of data and control between the filter, the host, and the various buffers involved.

This chapter discusses the design of the data paths and control facilities which implement data delivery.

- The data path section discusses a range of designs that are appropriate for different host architectures and different performance goals. The discussion begins with a design that requires a great deal of host support and evolves toward a design that minimizes host support of the network interface. Different designs might be found on the same network for machines with different capabilities; because all incorporate the filter, all of the filter protocols would still work.
- The control section discusses the programming support, other than filter code, required in the interface.

6.2 DATA PATHS

Common Features

All of the designs have certain features in common; these features are some form of transceiver, a two-part host interface, and the filter machine.

The transceiver connects the network interface to the medium. It is shown as a box labeled "Transceiver" and contains all of the analog circuits, isolation, and appropriate serialization, deserialization, and repeater logic. The transceiver can use either a circular or bus medium; the differences between the two systems do not affect the rest of the design.

The interface to the host has two separate paths: one for data transfers, and one for status, control, and interrupt signals. The data transfer section is either a shared memory interface, shown as a box labeled "Bus Window," or a channel or DMA type interface, shown as a box labeled "DMA." The best choice depends on the host architecture. The status, control, and interrupt signals are shown as a box labeled "SCI". In general, the logic for both paths is highly host dependent.

The filter machine is shown as a box labeled "Filter" for the machine itself, and a box labeled "Filter Memory" for the filter specifications. The actual filter program, exclusive of the filter specifications, may be implemented either as part of the filter memory or as a separate memory. The following designs assume a single memory which is initialized by some agency other than the filter machine itself.

Notation

Control paths are omitted in the interests of clarity. Busses are shown as bolder lines. Where multiport memory is shown, the access paths are numbered by priority, with 1 being highest.

Designs

Figure 6-1 shows a minimal interface.

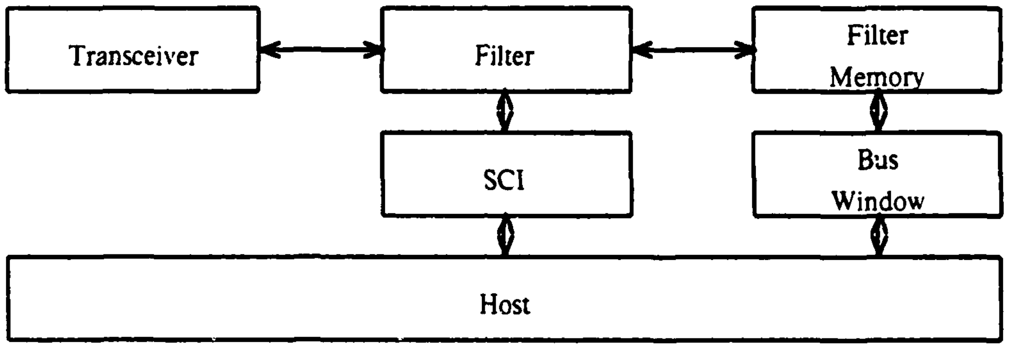


Figure 6-1: Data paths for minimal interface

This structure is similar to existing microprocessor-based designs. This design assumes that the filter machine code is modified to perform additional tasks. These tasks are copying arriving packets into a section of filter memory, rudimentary buffer management, and generation of signals to drive the SCI logic. This extra duty would require either slightly faster processor logic or a medium slower than the target 10 Mbps. In any case, the filter program is complicated by the necessity to multiplex filtering with host requests.

The minimal design assumes that the host is responsible for maintaining the filter data base and performing all protocol tasks not performed by the filter. Even so, this design would allow the host to benefit from the triage performed by the filter, prompt acks, and automatic rejection of duplicate packets, packets with transmission errors, etc. This design might be appropriate for less capable hosts in a heterogeneous network.

Figure 6-2 shows a modified version of the minimal interface. In this design the arriving packets are stored in packet buffers by separate DMA logic. This DMA controller is very simple: basically it is a loadable counter. The host accesses packet data as before, but must access filter memory either through the SCI logic or by having the filter machine copy memory through packet buffers. This design allows full speed operation without modification. The host must still process all packets with "interesting" data.

Figure 6-3 shows a fully functional design which is functionally equivalent to the model interface proposed in chapter 4. This interface differs from the previous designs in that it incorporates a microprocessor and uses a buffer memory to store both raw packets and connection rings. Because of the microprocessor and its EPROM and RAM, the network interface is capable of all protocol processing, as well as interface initialization.

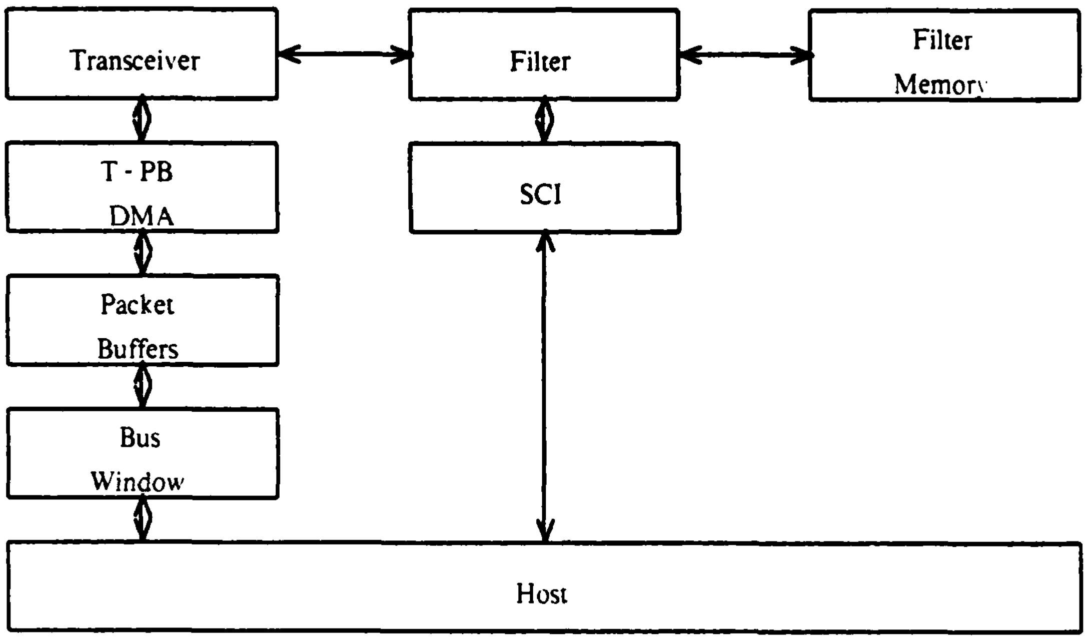


Figure 6-2: Data paths for full speed interface

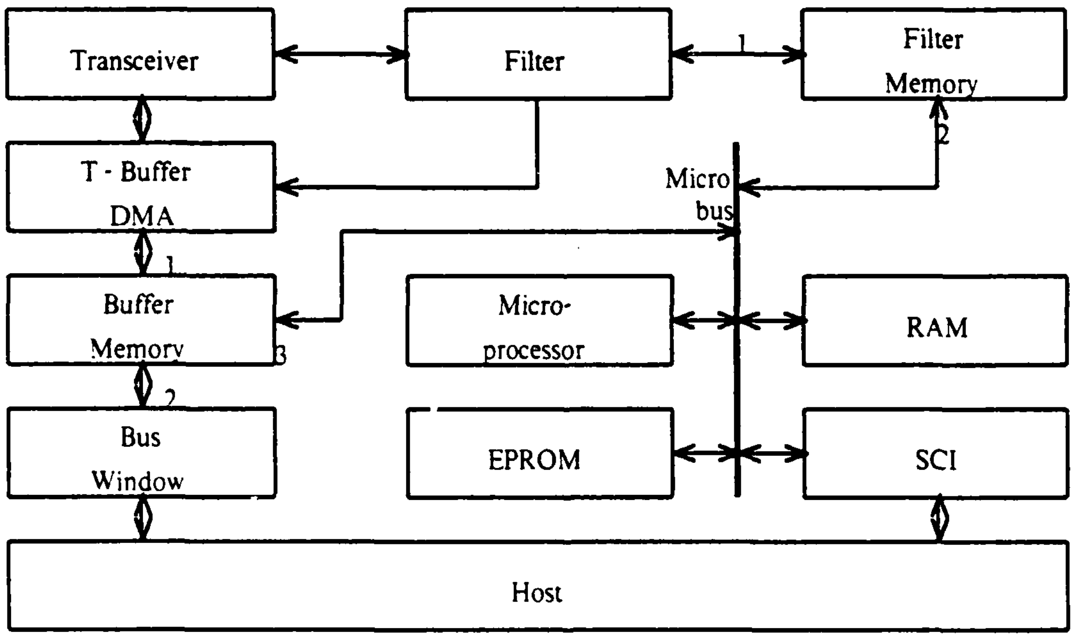


Figure 6-3: Data paths for microprocessor interface

The microprocessor interface can also maintain the connection-data rings in the host by replacing the bus window with a channel or DMA logic and adding additional control lines from the filter.

6.3 CONTROL

Magnitude of Task

Before discussing the programming support in the network interface, it is worthwhile to estimate the magnitude of the programming task based on existing protocol implementations. These estimates help to resolve the question of whether the interface microprocessor is necessary or whether it is practical to attempt to execute all filter code in the filter machine.

Table 6-1 lists the sizes of Network Control Programs (NCPs) as reported in [Postel 74]. Table IP/TCPIMPLSIZES shows the reported sizes of various IP/TCP implementations. The IP/TCP implementations are probably much less comparable due to subsetting. In particular, the ALTO and Stanford TCP project entries are designed for use in very small and restricted environments.

Table 6-1: NCP implementation sizes

System	Code (Kbits)	Table (Kbits)	Buffer (Kbits)	Total (Kbits)
TENEX	144	54	144	342
MULTICS	792	-	110	902
360/75	416	424	-	840
360/91	120	40	40	200
370/158	280	96	-	376
370/145	-	-	-	800
Burroughs 6700	720	288	-	1008
TIP	96	32	64	196
ANTS	64	8	4	76
DEC10	216	216	-	432
BKY	96	120	16	232

Based on these sizes, it seems likely that protocol software could be added to the filter machine on the basis of memory size. However, it also seems likely that the limited instruction set would make writing this software a formidable task. Given the low cost of microprocessors, and the availability of software development tools, it seems that moving these functions into the filter could only be justified for a very high production run of network interfaces. Even in that case, the diagnostic ability of the microprocessor or the availability of microprocessor peripheral chips, EPROMS, etc., might justify its inclusion.

Software Tasks in the Microprocessor

Since the microprocessor software can receive deferred packets at a very low level, it must include all of the conventional layers of protocol processing with added input points to accommodate deferred packets from different parts of the filter code. Thus the interface microprocessor software must start with a fairly standard set of front-end protocol processing routines. To be sure, our goal is to eliminate most of the usage of these routines; however, they must be present.

Table 6-2: IP/TCP implementation sizes

Implementor	Program	Language	Size (Kbits)	CPU
BBN TOPS-20	Multinet	assembly	263	PDP 10,20
[Lynn 82]	IP	"	134	"
	TCP	"	207	"
	Configuration data		32	"
	(Buffers and table space are extra)			
Stanford TCP	TCP	BCPL	192	PDP 11
Project	TCP	assembly	19	PDP 11
[Cerf 80]				
MIT (Clark)	Minimal	BCPL	48	ALTO
[Postel 81e]	IP/TCP/TELNET			
MIT MULTICS	IP	-	191 code	-
[Postel 81e]	TCP	-	324 code	-
BBN	IP/TCP/1822	SPL	64 code	HP 3000
[Postel 81e]			160 data	
EDN-UNIX	IP/TCP	-	221 code	VAX
[Cain 82]			194 data	

The retransmission parameters used in these routines must be carefully set. The rapid response possible with prompt acknowledgments makes severe cases of the so called "silly window syndrome" possible. The root cause of this potential problem is the speed of acknowledgment feedback to a sending process. If this feedback is used indiscriminately, the average transmission size can become quite small as the source of octets gets acknowledgments almost as fast as it can generate data. This type of behavior is desirable where reducing latency is of primary importance; however, it is wasteful in many environments, e.g., FTP.

The remaining software tasks of the microprocessor are concerned with maintaining the filter data base and user interfacing. These tasks should be much less memory consuming than the protocol processing software.

7. SUMMARY AND CONCLUSIONS

7.1 RESULTS

The main conclusion of this report is that it is possible to add a fairly simple processing element, the filter, to a microprocessor-based network interface and create an interface that can, in most cases, perform all levels of protocol processing as the packet arrives. The filter and microprocessor form a processing hierarchy: the filter processes a small number of distinct packet formats which make up the majority of the packet population; what the filter cannot process, it passes on to the microprocessor. The delay and throughput limitations due to the time consumed in executing protocol software are nearly eliminated.

- The filtering idea proved to be reasonable for IP and TCP, in addition to special protocols designed to make filtering easy. There is no reason to suspect that other protocols, such as X.25 or the new Xerox family, could not be supported as easily.
- The real-time constraints in this report were admittedly set to match existing component speeds and the popularity of 10 Mbps media. However, the demonstration also sets an upper bound on the number of computations per unit of arriving data which are required to achieve real-time processing. For the proposed filter design, the ratio is 1 memory operation for every 4 bits of arriving data, and 1 16-bit arithmetic operation for every 2 bits. For a 10 Mbps medium, this results in a 400 ns memory and a 5 Mip, 16-bit machine. In the case of IP this bound is quite loose; for TCP, the bound appears to be quite close.
- The filter has several practical advantages. It is a memory-oriented design and as such will be able to enjoy the benefits of faster, denser, and cheaper memories. The signal interfaces to the filter per se are such that it could easily be built as a single custom VLSI chip without pinout problems. A VLSI filter does not have to drive any media rate signals off chip.

A secondary set of results concerns the use of new features designed to exploit the real-time capability of the filter. The most promising new feature is the prompt acknowledgment, which is an acknowledgment which is piggybacked on the media allocation of the transmission it acknowledges. The prompt acknowledgments of the filter interface differ from previous systems in that they interface with all protocol levels, rather than simply the link level.

Prompt acknowledgments preserve the conventional protocol-layering model if they are treated as "shorthand" versions of packets which could just as well be sent as conventional transmissions (if not as efficiently). The semantics of prompt acknowledgments are tied to the semantics of the equivalent expanded acknowledgment packets.

Prompt acknowledgments do not replace the need for regular acknowledgment transmissions; instead they create a very efficient parallel mechanism which usually, but not always, avoids the need for the more costly acknowledgment. Thus an interface which uses prompt acknowledgments can talk to an interface which does not.

This preservation of compatibility with nonfiltering interfaces, end-to-end acknowledgments, and conventional layering carries a price: Filtering interfaces can never reduce the amount of protocol code, instead the amount of protocol code is increased by the duplicate functionality that must be supported, and by new sensitivities to timeout, packeting, bufferir g, and other policies.

7.2 SUGGESTIONS FOR FUTURE WORK

The least attractive features of the filter discussed in Chapter 5 are associated with its programming. In particular, the filter code is inconveniently wide, difficult to write, and lacking in the machine features of more general machines. In a sense, this is desirable in that it suggests that the filter machine is in some sense minimal. However, a VLSI implementation would destroy this minimality argument. A better programming environment would enable a more flexible and usable interface.

The most important architectural issue remaining is the design of better methods for incorporating communications, or for that matter any intelligent front or back end, to the process structure of the host. Efficient data paths, whether shared memory, DMA, or channels are slowed by inefficient control as embodied in interrupt, context swap, and scheduling facilities of the host OS. A special computing element which can handle all of these control functions for the most common situations might be the answer. A DCS style of multiple process computation, in which the user process scheduling decisions are based on message arrival, message transmission, and timeouts might be a good starting point.

7.3 IMPLICATIONS OF THE FILTER ARCHITECTURE

The advantages of local networks derive from the exploitation of cheap bandwidth and cheap processing power. These resources are so cheap that it makes no sense to worry about their efficient use per se. The potential of the filter interface is not that it can use these resources more efficiently, but rather that it can make high-level communications service, for example TCP connections, into another very cheap resource.

The qualitative advantage of this cost reduction is that it expands the range of applications which can be implemented using the higher level protocol, and hence reduces the need for spartan performance oriented protocols. If a protocol with full features costs the same as a simpler protocol, there is no need to divert resources to the construction, maintenance, and training costs associated with multiple protocols. The least tangible, and yet probably the most important effect, is that processes have one less way to be incompatible.

Of course, some applications require services that cannot be made to fit into the TCP mold. For example, reliability through retransmission is anathema to some real-time applications, notably voice.

A promising new approach made possible by the filter is to create a "new" protocol by adding features to a TCP base, rather than creating a new protocol from scratch. For voice, one could imagine a new protocol which added a timer driven update to the receive sequence space which in effect acknowledges data after a certain period whether it arrived or not. For such an approach to succeed, the state of the connection must be explicit and available to the additional features, and perhaps more importantly users of the standard service should be protected from the experimentation of others.

GLOSSARY

Note: Terms labeled "*" are specific to this report.

1822	1822 refers to BBN technical report 1822, which defines the protocol used between the host and IMP in the ARPANET. The term is used to mean the protocol.
2900	A family of bit-slice components available from AMD, National Semiconductor, and others.
6800	An 8-bit NMOS microprocessor manufactured by Motorola.
8080	An 8-bit NMOS microprocessor manufactured by Intel.
8X300	A bipolar microprocessor manufactured by Signetics.
ACK	An acknowledgment. Also a TCP control flag which indicates that the TCP segment carries a valid acknowledgment sequence value.
ALU	Arithmetic and Logic Unit - refers to the part of a computer that contains the circuits that perform arithmetic operations.
ARPANET	The ARPANET is an operational, resource sharing, host-to-host network linking a wide variety of computers at DoD facilities and non-DoD research centers in the continental United States, Hawaii, Norway, and England [Roberts 70, McQuillan 77].
BATNET	A local network with a bus structure built at Battelle-Northwest Labs [Gerhardstein 78]. See page 29.
broadcast	Distribution to all addresses in a network.
Cambridge loop	A local network with a loop architecture built at the Computer Lab at the University of Cambridge in England [Wilkes 79, Wilkes 80]. See page 54.
CATV	Community Area TeleVision - cable TV.
Chaosnet	A bus network built at the MIT AI lab [Shoch 79]. See page 31.
CID *	Connection ID - a term used to describe the parts of a packet that uniquely specify a particular connection. A CID is usually a combination of a source and a destination address.

CMOS	Complementary MOS - CMOS circuits include both NMOS and CMOS devices on a single chip. CMOS offers slightly higher performance, extremely low power consumption, and high noise immunity. However it requires a more complicated manufacturing process and has inherently lower density. CMOS components have typical gate delays in the 10 ns range [Myers 80].
Consortium Ethernet	The new Ethernet standard for a 10 Mbps bus proposed by Xerox, DEC, and Intel Corps. [DEC 80, CD 1980]. See Ethernet and page 35.
CRC	Cyclic Redundancy Check - CRC values are the remainder when a string of bits is divided by a specific polynomial. Several standard polynomials are defined. The CRC is usually formed using special hardware and synthetic division. CRCs are used in a manner similar to checksums.
CSMA	Carrier Sense Multiple Access - an access control algorithm in which a station wishing to transmit listens for transmissions in progress and waits for an idle medium before transmitting.
CSR *	Conditional Shift Register - a register in the filter machine proposed in Chapter 4.
DCS	Distributed Computer System - a network built at UC Irvine [Farber 73, Mockapetris 80] consisting of minicomputer hosts and a ring network built using ring interfaces (RIs) [Loomis 73]. See Section 3.2.
DLCN	The Distributed Loop Computer Network is a loop network built at the Ohio State University [Liu 78]. See page 53.
DMA	Direct Memory Access - used to refer to peripherals that access memory directly, i.e., without CPU intervention.
ECL	Emitter Coupled Logic - ECL is the fastest technology available in the marketplace today; ECL achieves gate delays in the .2-2 ns range [Myers 80].
ENET	[West 77] discusses design decisions behind ENET. ENET is one of the two networking plans discussed. The two designs are ENET (Expensive network) and CNET (Cheap network). See page 30.
EPROM	Erasable Programmable Read Only Memory - a ROM that can be electrically programmed. EPROMS are high density, but slower than PROMS. The most common EPROMS can be erased with UV light and reprogrammed.
Ethernet	The Ethernet is the best known bus network, and was originally developed at the Xerox Palo Alto Research Center. Metcalfe 76] contains basic information about the theory of operation and motivations for the Ethernet. The Ethernet patent [Metcalfe 77], and [Crane 80] contain detailed information regarding interface implementation and other practical considerations. See page 25.

	The original Ethernet, sometimes called the research Ethernet, is to be superseded by the new 10 Mbps Ethernet designed by Xerox, DEC, and Intel Corps. See *consortium Ethernet*.
FDMA	Frequency Division Multiple Access - multiplexing by combining signals with different carrier frequencies.
FIFO	First In First Out - often used as a name for a memory that exhibits first in, first out behavior.
FIN	A TCP control bit occupying one sequence number, which indicates that the sender will send no more data or control occupying sequence space.
FPLA	Field Programmable Logic Array - a PLA with fusible links that can be programmed much like a PROM. A typical FPLA (Signetics 8S100) has 16 inputs, 48 terms, and 8 outputs.
FR *	Filter Register - the FR is a register in the filter machine which holds the address of the next N-way table for control filtering, as opposed to CID recognition.
FSM	Finite State Machine.
FTP	File Transfer Protocol - originally a protocol used to transfer files between machines on the ARPANET. Now used generically for such a protocol.
GMAD	A CATV based bus network built at the General Motors Assembly Division and described in [Smith 79]. See page 33.
HDLC	High-level Data Link Control - a link level standard.
HYPERchannel	The HYPERchannel is a 50 Mbps bus network available commercially from Network Systems Corp. [Thornton 75, Donnelley 79, Thornton 79]. See page 33.
IC	Integrated Circuit - a combination of interconnected circuit elements inseparably associated on or within a continuous substrate; a "chip".
IDA	The IDA network is a proposed 24-bit parallel loop [Bliss 78]. See page 58.
IMP	Interface Message Processor - a minicomputer based host interface and message switch used in the ARPANET.
IP	Internet Protocol - the lower level protocol in the DoD Standard IP/TCP pair [Postel 81b]. "Internet" is also the name used by Xerox for its transport level protocols for the new Ethernet system [Xerox 81b], although the new protocols are completely different from the DoD IP.

LED	Light Emitting Diode - a semiconductor junction which emits light and hence is used to build indicators, opto-isolators, etc.
LNI	Local Network Interface - LNI refers to members of a family of local network interfaces, all of which use a ring medium and token passing. The first LNI was designed at UC Irvine and is in use at MIT [Mockapetris 77]. Subsequent versions were created by MIT and Network Systems. For the UCI and MIT LNI see page 41; for the Network Systems version see page 42.
LSI	Large Scale Integration.
MAR	Memory Address Register - the register which holds the address of the next main memory operation.
MDR	Memory Data Register - the register which holds the data which is being transfered to or from memory.
Mitre network	A CATV based bus network described in [Hopkins 79, Holmgren 79]. See page 32.
MOS	Metal-Oxide Semiconductor - MOS is the basic silicon technology used to build all VLSI and much of the LSI available today. Most microprocessors and the denser memories are built using MOS. MOS gate delays in commercially available products range from 100 ns (or greater) to about 1 ns [Myers 80]. See CMOS, NMOS, and PMOS for additional information.
MSI	Medium Scale Integration.
multicast	Distribution to a selected subset of all addresses. See page 150.
NACK	Negative acknowledgment - a signal or acknowledgment which signifies that a particular transmission has failed.
NBS network	A bus network built at the National Bureau of Standards, and described in [Carpenter 78]. See page 30.
NCP	Network Control Protocol - the process to process protocol of the ARPANET.
Newhall	The first token loop network [Farmer 69]. See page 40.
NMOS	N-channel MOS - NMOS was the second major MOS technology and has replaced PMOS largely because of its better speed. The majority of second generation microprocessors (e.g., 8080, 6800) and newer microprocessors (68000, 8086) are built with NMOS, as are almost all high-density memories. NMOS gate delays are in the 1-20 ns range [Myers 80].
octet	An 8-bit byte.

PC	Program Counter - a register which holds the address of the next instruction to be executed.
Pierce loop	Pierce loops were one of the pioneering efforts in building slotted loop networks [Kropfl 72, Coker 72, Pierce 72a, Pierce 72b]. See page 41.
PLA	Programmed Logic Array.
PMOS	P-channel MOS - the original form of MOS which has largely been superseded by NMOS and CMOS. PMOS was used for the first calculator and microprocessor chips, but is rarely used in new designs. PMOS gate delays are 100 ns and up [Myers 80].
port	A portion of a socket which specifies which logical input or output channel of a process is associated with the data.
PRIMENET	A ring network built by Prime Computer Corp described in [Farr 77]. See page 56.
PROM	Programmable Read Only Memory - a ROM that can be programmed by blowing fusible links.
prompt acknowledgment *	Used in this report to refer to an acknowledgment transmitted within the same medium allocation as the packet it acknowledges. See page 105.
PSH	A TCP control bit occupying no sequence space, indicating that the data in the segments should be pushed through to the receiving user.
QBUS	The QBUS is the processor bus used in the LSI versions of DEC's PDP-11 series. See UNIBUS and [DEC 79].
RAM	Random Access Memory - a memory in which all contents are equally accessible.
RI	Ring Interface - the network interface used in the DCS system to implement a token controlled ring [Loomis 73]. See page 41.
RS-232	A bit serial interface standard used primarily for interfacing terminals to computers.
RST	A TCP control bit (reset) occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can decide, based on the acknowledgment and sequence values in the segment, whether or not to honor the RST. **In no case should the receipt of a RST segment result in the transmission of a RST segment in response.**
S-100	A widespread bus standard for microprocessors.

Salplex	A bus network for use in trucks and other large vehicles. Described in [Smith 80a]. See page 36.
SDLC	Synchronous Data Link Control - an IBM link level protocol [IBM 74].
segment	IP/TCP terminology for a packet.
Spider	A loop network built at Bell Labs and described in [Fraser 75]. See page 41.
SR *	Shift Register - a register in the filter machine which holds arriving data.
SSI	Small Scale Integration.
SYN	A TCP control bit (synchronize), occupying one sequence number, used during connection initiation to indicate where sequence numbering will start.
TAC	Terminal Access Controller. TACs are second generation TIPs, and include support for IP/TCP.
TCP	Transmission Control Protocol - a DOD standard protocol for communications between processes [Postel 81a]. See page 78.
TDMA	Time Division Multiple Access - multiplexing by periodic sharing of the medium.
TDR	Time Domain Reflectometer an instrument that determines the length of a piece of coax by measuring the time a pulse takes to be reflected back to the TDR.
TIP	Terminal Interface Processor - an IMP that can also support terminal interface to the ARPANET.
Toshiba loop	A loop network described in [Okuda 78]. See page 56.
TRW ring	A ring network described in [Blauman 79]. See page 57.
TTL	Transistor Transistor Logic - TTL refers to several forms of bipolar logic used to fabricate MSI and some LSI devices. The fastest form of TTL, advanced Schottky or ASTTL has a gate delay of approximately 1.5 ns; a more common form of TTL, low-power Schottky or LS, has a gate delay of approximately 10 ns, but uses only 1/10th as much power [Myers 80].
UNIBUS	The UNIBUS is the processor bus used in most of the DEC PDP-11 medium to large size minicomputers. See QBUS and [DEC 79].
URG	A TCP control bit (urgent), occupying no sequence space, used to indicate to the receiving user that urgent processing should be performed on data octets up to the octet designated by the urgent pointer.

VLSI	Very Large Scale Integration.
X.25	A CCITT standard transport level service specification [CCITT 77].
Z80	An 8-bit MOS microprocessor, similar to the 8080, manufactured by ZILOG.

REFERENCES

[Abramson 70] Abramson, N., "Another alternative for computer communication," in *AFIPS Conference Proceedings, Volume 37: Fall Joint Computer Conference*, Houston, Tex., November 1970, pp. 695-702.

[Abramson 73a] Abramson, N., "Packet switching with satellites," in *AFIPS Conference Proceedings, Volume 42: National Computer Conference*, New York, June 1973, pp. 695-702.

[Abramson 73b] Abramson, N., "The ALOHA system," in N. Abramson and F. Kuo (eds.), *Computer Communications Networks*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[Acree 76] Acree, J., and Lynch, A., "Ring architecture supports distributed processing," *Data Communications*, March/April 1976, 51-55.

[Agrawala 77] Agrawala, A. K., R. M. Bryant, and J. Agre, *Analysis of an Ethernet-like Protocol*, Technical Report TR-522, University of Maryland, Department of Computer Science, April 1977.

[Agrawala 78] Agrawala, A. K., J. R. Agre, and K. D. Gordon, "The slotted ring vs. the token controlled ring: A comparative evaluation," in *Proceedings of the Computer Software and Applications Conference*, 1978, pp. 674-679.

[Anderson 75] Anderson, G. A., and E. D. Jensen, "Computer interconnection structures: Taxonomy, characteristics, and examples," *Computing Surveys* 7, December 1975.

[Anderson 79] Anderson, R. J., "Local data networks - traditional concepts and methods," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 127-149.

[Arvind 78] Arvind, and K. P. Gostelow, *Dataflow Computer Architecture: Research and Goals*, Technical Report 113, University of California, Irvine, Department of Information and Computer Science, February 1978.

[Ashenhurst 75] Ashenhurst, R. L., and R. H. Vonderohe, "A hierarchical network," *Datamation* 21 ([illegible] February 1975, 40-44.

[Babic 77] Babic, G., and T. L. Ming, "A performance study of the Distributed Loop Computer Network (DLCN)," in *Proceedings of the Computer Networking Symposium*, National Bureau of Standards, Gaithersburg, Md., December 15, 1977, pp. 66-76.

[[illegible] 76] Ball, J. E., J. Feldman, J. R. Low, R. Rashid, and P. Rovner, "RIG, Rochester's Intelligent Gateway: System overview," *IEEE Transactions on Software Engineering* SE-2 (4), December 1976, 321-328.

[Ball 78] Ball, J., G. Williams, and J. Low, *Preliminary ZENO Language Description*, University of Rochester, Computer Science Department, TR41, December 1978.

[Balzer 71] Balzer, R. M., "Ports - a method for dynamic interprogram communication and job control," *AFIPS Conference Proceedings, Volume 38: Spring Joint Computer Conference*, Atlantic City, N. J., May 1971, pp. 485-489.

[Baran 64] Baran, P., et al., *On Distributed Communication I-XI*, RAND Corporation Research Documents, August 1964.

[Bartlett 69] Bartlett, K. A., R. A. Scantlebury, and P. T. Wilkenson, "A note on reliable full-duplex transmission over half-duplex links," *Communications of the ACM* 12 (5), May 1969.

[Bartlett 78] Bartlett, J. F., "A nonstop operating system," in *Proceedings of the Eleventh Hawaiian International Conference on System Sciences*, 1978, pp. 103-117.

[Belsnes 76] Belsnes, D., "Single message communication," *IEEE Transactions on Communication* COM-24 (2), February 1976, 190-194.

[Belton 74] Belton, R. C., and J. R. Thomas, "The United Kingdom Post Office packet switching experiment," in *Proceedings of the International Switching Symposium*, Munich, September 1974.

[Biba 79] Biba, K. J., and J. W. Yeh, "FordNet: A front-end approach to local computer networks," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 199-215.

[Binder 75] Binder, R., N. Abramson, F. Kuo, A. Okinaka, and D. Wax, "ALOHA packet broadcasting--A retrospect," *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 203-215.

[Blauman 79] Blauman, S., "Labeled slot multiplexing: A technique for a high speed, fiber optic based, loop network," in *Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks*, Lawrence Berkeley Laboratory, August 1979, pp. 309-321.

[Blauuw 70] Blauuw, G. A., "Hardware requirements for the fourth generation," in F. Gruenberger (ed.), *Fourth Generation Computers: User Requirements and Transaction*, Prentice-Hall, Englewood Cliffs, N. J., pp. 155-168, 1970.

[Bliss 78] Bliss, B., "IDANET, an application," in *Proceedings of the Third Conference on Local Computer Networks*, University of Minnesota, October 1978.

[Bochmann 76] Bochmann, G. V., and R. J. Chung, *A Formalized Specification of HDLC Classes of Procedures*, Technical Report 265. University of Montreal, Dept. d'I R. O., 1976. Also in *Proceedings of the National Telecommunications Conference*, IEEE, 1977, and reprinted in W. Chu (ed.), *Advances in Computer Communications and Networking*, Artech House, Dedham, Mass., 1979, pp. 519-530.

[Boggs 79]	Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe, Pup: An internetwork architecture, Xerox PARC working draft, May 1979.
[Borgerson 76]	Borgeson, B. R., "The viability of multimicroprocessor systems," *Computer*, January 1976, 26-30.
[BSTJ 81]	Issue on No. 4 Electronic Switching System, *Bell System Technical Journal* 60 (6), part 2, July-August 1981.
[Bunch 80]	Bunch, S. R., and J. D. Day, "Control structure overhead in TCP," in *Proceedings Trends & Applications: 1980, Computer Network Protocols*, May 1980, pp. 121-127.
[Burchfield 75]	Burchfield, J., R. Tomlinson, and M. Beeler, "Functions and structure of a packet radio station," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 245-251.
[Burkhard 76]	Burkhard, W. A., "Hashing and trie algorithms for partial match retrieval," *ACM Transactions on Database Systems*, 1 (2), June 1976, 175-187.
[Cain 82]	Cain, E., personal communication, May 1982.
[Campbell 74]	Campbell, G. H., K. Fuchel, S. L. Padwa, and N. F. Schumberg, *BROOKNET-A High Speed Computer Network*, Brookhaven National Laboratory, Upton, New York, Technical Report BNL-19365, September 1974.
[Carpenter 78]	Carpenter, R. J., J. Sokol, and R. Rosenthal, "A microprocessor-based local network node," in *Proceedings Compcon Fall 78*, Washington, D. C., September 1978, pp. 104-109.
[Carr 70]	Carr, C. S., S. D. Crocker, and V. G. Cerf, "Host-Host communication protocol in the ARPA network," in *AFIPS Conference Proceedings, Volume 36: Spring Joint Computer Conference*, Atlantic City, N. J., May 1970, pp. 589-597.
[Casey 77]	Casey, L. M., *Computer Structures for Distributed Systems*, Ph.D. thesis, University of Edinburgh, December 1977.
[CCITT 77]	CCITT, "X.25 interface between data terminal equipment (DTE) and data circuit-terminating equipment (DCE) for terminals operating in the packet mode on public data networks," in *CCITT Sixth Plenary Assembly Orange Book*, volume VIII.2 on Public Data Networks, published by the International Telecommunication Union, Geneva, 1977.
[CD 80]	"Three companies develop local network specifications," *Computer Design*, pp. 40-44, July 1980.
[Cerf 80]	Cerf, V. G., "Final report of the Stanford University TCP project," IEN 151, USC/Information Sciences Institute, April 1980.

[Christensen 77] Christensen, G. S., "Data truck contention in a HYPER channel network," in *Proceedings of the Conference on a Second Look at Local Computer Networking*, University of Minnesota, October 1977.

[Clark 77a] Clark, D. D., "A contention ring network," M.I.T. Laboratory for Computer Science, Local Network Note No. 11, September 16, 1977.

[Clark 77b] Clark, D. D., "Comparison of TCP and DSP," M.I.T. Laboratory for Computer Science, Local Network Note No. 7, April 28, 1977.

[Clark 77c] Clark, D. D., "Revision of DSP specification," M.I.T. Laboratory for Computer Science, Local Network Note No. 9, June 17, 1977.

[Clark 78a] Clark, D. D., I. Grief, B. Liskov, and L. Svobodova, *Semantics of Distributed Computing*, Progress Report of the Distributed Systems Group, M.I.T. Laboratory for Computer Science, 1978.

[Clark 78b] Clark, D. D., K. T. Pogran, and D. P. Reed, "An introduction to local networks," in *Proceedings of the IEEE* 66 (11), November 1978, 1497-1516.

[Clipsham 76] Clipsham, W. W., F. E. Glave, and M. L. Narraway, "DATAPAC network overview," in *Proceedings of the International Conference on Computer Communication*, Toronto, August 1976, pp. 131-136.

[Codd 62] Codd, E. F., "Multiprogramming," in F. L. Alt and M. Rubinoff (eds.), *Advances in Computers*, Vol. 3, Academic Press, 1962, pp. 77-153.

[Cohen 75] Cohen, E., and D. Jefferson, "Protection in the HYDRA operating system," in *Proceedings of the Fifth Symposium on Operating Systems Principles*, Austin, Tex., November 1975, as *ACM Operating Systems Review* 9 (5), pp. 141-160.

[Cohen 76] Cohen, D., *Specifications for the Network Voice Protocol*, USC/Information Sciences Institute, ISI/RR-75-39, March 1976.

[Coker 72] Coker, C. H., "An experimental interconnection of computers through a loop transmission system," *Bell System Technical Journal* 51 (6), July-August 1972, 1167-1175.

[Cole 71] Cole, G. D., "Performance Measurements on the ARPA Computer Network," University of California, Los Angeles, Computer Science Department, 1971.

[Cotton 79] Cotton, I. W., "Technologies for local area computer networks," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 25-44.

[Counterman 76] Counterman, W. A., "IDANET," in *Proceedings of the University of Minnesota Conference on Local Networks*, September 1976.

[Crabtree 78] Crabtree, R. P., "Job networking," *IBM Systems Journal* 17 (3), 1978, 206-220.

[Crane 80] Crane, R. C., and E. A. Taft, "Practical considerations in Ethernet local network design," in *The Ethernet Local Network: Three Reports*, Xerox PARC Technical Report CSL-80-2, February 1980. Shorter version also presented at the Hawaiian International Conference on System Sciences, January 1980.

[Crocker 77] Crocker, D. H., et al., "Standard for the format of ARPA network text messages," ARPA Network RFC No. 733, NIC 41952, USC/Information Sciences Institute, November 21, 1977.

[CSDL 70] "STS software development (Study task 5)," M.I.T. Charles Stark Draper Laboratories, July 1970.

[Dalal 76] Dalal, Y. K., *A Distributed Algorithm for Constructing Minimal Spanning Trees in Computer Communication Networks*, Technical Report 111, Stanford University, Digital Systems Laboratory, 1976.

[Dalal 77] Dalal, Y. K., *Broadcast Protocols in Packet Switched Computer Networks*, Ph.D. thesis, Technical Report 128, Stanford University, Digital Systems Laboratories, April 1977.

[Dalal 78] Dalal, Y. K., and R. M. Metcalfe, "Reverse path forwarding of broadcast packets," *Communications of the ACM* 21 (12), December 1978, 1040-1048.

[Danet 76] Danet, A., R. Depres, A. LesRest, G. Pichon, and S. Ritzenthaler, "The French public packet switching service: The TRANSPAC network," *Proceedings of the International Conference on Computer Communication*, Toronto, August 1976, pp. 251-260.

[DATAPAC 76] The Computer Communication Group, *Standard Network Access Protocol Specification (SNAP)*, Datapac manual, 1976.

[Davies 67] Davies, D. W., et al., "A digital communication network for computers giving rapid response at remote terminals," *ACM Symposium on Operating Systems Problems*, October 1967.

[Day 77] Day, J., "Offloading ARPANET protocols to a front end," CAC Doc 230, 1977.

[DEC 79] *PDP-11 Bus Handbook*, Digital Equipment Corporation, AR-EB-17525-20, 1979.

[DEC 80] "The Ethernet, a local area network, data link layer and physical layer specifications," published jointly by Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, September 1980.

[DeMarines 76] DeMarines, V. A., and L. W. Hill, "The cable bus in data communication," *Datamation* 22 (8), August 1976, 89-92.

[Dennis 66] Dennis, J. B., and E.C. Van Horne, "Programming semantics for multiprogrammed computations," *Communications of the ACM* 9 (3), March 1966, 143-155.

[Deutch 79] Deutch, Debra P., "A suggested solution to the naming, addressing, and delivery problem for ARPAnet message systems," ARPA Network RFC 757, USC/Information Sciences Institute, September 1979.

[Dolotta 76] Dolotta, T. A., and J. R. Mashey, "An introduction to the Programmer's Workbench," in *Proceedings of the Second International Conference on Software Engineering*, October 1976, pp. 164-168.

[Donan 74] Donan, R. A., and J. R. Kersey, "Synchronous data link control: A perspective," *IBM Systems Journal* 13 (2), 1974, 140-162.

[Donnelly 78a] Donnelly, J. E., and J. W. Yeh, "Interaction between protocol levels in a prioritized CSMA broadcast network," in *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks*, San Francisco, August 1978, pp. 123-143.

[Donnelly 78b] Donnelly, J. E., and J. W. Yeh, "Simulation studies of round robin contention in a prioritized CSMA broadcast network," in *Proceedings of the Third Conference on Local Computer Networks*, University of Minnesota, October 1978.

[Earl 73] Earl, W. J., "Interfacing the DCS to the ARPANET," DCS memo 63, University of California, Irvine, Department of Information and Computer Science, March 1973.

[Elie 78] Elie, M., "Traffic control by latency in a packet switching," presented at the 1978 Flow Control Conference, Liege.

[Enslow 77] Enslow, P. E., "Multiprocessor organization - A survey," *ACM Computing Surveys* 9 (1), March 1977, 103-129.

[Fabry 74] Fabry, R. S., "Capability-based addressing," *Communications of the ACM* 17 (7), July 1974, 403-412.

[Farber 70] Farber, D. J., "A distributed computer system," University of California, Irvine, Department of Information and Computer Science, Technical Report 4, September 1970.

[Farber 72a] Farber, D. J., and F. R. Heinrich, "The structure of a distributed computer system-- the file system," in *Proceedings of the International Conference on Computer Communication*, October 1972, pp. 364-370.

[Farber 72b] Farber, D. J., and K. Larson, "The structure of a distributed computer system--The communication system," in *Proceedings of the Symposium on Computer Communications Networks and Teletraffic*, Jerome Fox, ed., Polytechnic Press, New York, April 1972, pp. 21-27.

[Farber 72c] Farber, D. J., "The state of computer network technology," *Datamation*, invited paper published in special network issue, 1972. Reprinted in *Tutorials in Computer Networks*, IEEE Press, First and Second Editions, 1978.

[Farber 72d] Farber, D. J., and K. C. Larson, "The structure of a distributed computing system--software," in *Proceedings of the Symposium on Computer Communications Networks and Teletraffic*, Jerome Fox, ed., Polytechnic Press, New York, April 1972, pp. 539-545.

[Farber 73] Farber, D. J., et al., "The distributed computing system," in *Proceedings Compcon Spring 73*, San Francisco, February 1973, pp. 31-34.

[Farber 74a] Farber, D. J., "Software considerations in distributed architectures," *IEEE Computer* 7 (3), March 1974, 31-35.

[Farber 74b] Farber, D. J., *Distributed data bases--An exploration*, Caine, Farber & Gordon, Inc., report, 1974.

[Farber 74c] Farber, D. J., "An overview of distributed processing aims," in *Digest of Papers, Compcon Fall 74*, September 1974, pp. 191-193.

[Farber 75] Farber, D. J., "A ring network," *Datamation*, 21 (2), February 1975, 44-46.

[Farber 76a] Farber, D. J., and J. Pickens, "The overseer - A communication attribute," in *Proceedings of the International Conference on Computer Communication*, 1976.

[Farber 76b] Farber, D. J., and K. Larson, "Network security via dynamic process renaming," in *Proceedings of the Fourth Data Communications Symposium*, 1976.

[Farber 77] Farber, D. J., and P. Baran, "The convergence of computing and telecommunication systems," *Science* 195 (4283), March 1977, 1166-1170.

[Farmer 69] Farmer, W. D., and E. E. Newhall, "An experimental distributed switching system to handle bursty computer traffic," in *Proceedings of the ACM Symposium on Problems in the Optimization of Data Communication Systems*, Pine Mountain, Ga., October 1969, pp. 31-34.

[Farr 77] Farr, W., "Proposed distributed computer network," preliminary specification PE-T-307, Prime Computer Corporation, February 1977.

[Feldman 76] Feldman, J. A., *A programming methodology for distributed computing (among other things)*, University of Rochester, Computer Science Department, TR9, Rochester N. Y., September 1976.

[Feldman 77a] Feldman, J. A., *Synchronizing Distant Cooperating Processes*, University of Rochester, Computer Science Department, TR26, Rochester N. Y., October 1977.

[Feldman 77b] Feldman, J. A., and G. J. Williams, *Some Comments on Data Types*, University of Rochester, Computer Science Department, TR28, Rochester N. Y., 1977.

[Feldman 78] Feldman J. A., J. R. Low, and P. D. Rovner, "Programming distributed systems," in *Proceedings of the 1978 Annual ACM Conference*, vol. 1, Washington D.C., December 1978, pp. 310-316.

[Feldman 79] Feldman, J. A., "High level programming for distributed computing," *Communications of the ACM* 22 (6), June 1979, 353-367.

[Fletcher 73] Fletcher, J. G., "The octopus computer network," *Datamation* 19 (4), April 1973, 58-63.

[Fletcher 78] Fletcher, J. G., and R. W. Watson, "Mechanisms for reliable timebased protocol," *Computer Networks* 2 (45), September/October 1978, 271-290.

[Forsdick 77] Forsdick, H. C., R. E. Schantz, and R. H. Thomas, *Operating Systems for Computer Networks*, Bolt Beranek and Newman Inc., Technical Report No. 3614, July 1977.

[Fortune 77] Fortune, P. J., W. P. Lidinsky, and B. R. Zelle, "Design and implementation of a local computer network," ILN Note No. 11, Argonne, April 1977. (Also in *Conference Record, International Conference on Communication*, Chicago, 1977.)

[Fralick 75a] Fralick, S. C., and J. C. Farrett, "Technological considerations for packet radio networks," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 233-234.

[Fralick 75b] Fralick, S. C., D. H. Brandin, F. F. Kuo, and C. Harrison, "Digital terminals for packet broadcasting," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 253-261.

[Franck 79] Franck, A., and P. C. Patton, "Some architectural and system implications of local computer networks," in *Proceedings Compcon Spring 79*, San Francisco, February-March 1979, pp.272-275.

[Frank 75] Frank, H., I. Gitman, and R. Van Slyke, "Packet radio system - Network considerations," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 217-231.

[Frank 76a] Frank, H., I. Gitman, and R. Van Slyke, *Recent Advances in Ground Packet Radio Systems*, Network Analysis Corporation, Sixth Semiannual Technical Report for the Project: Local, Regional and Larger Scale Integrated Networks, Volume 2, February 1976.

[Frank 76b] Frank, H., I. Gitman, and R. Van Slyke, *Local and Regional Communication Networks--Technologies and Architectures*, Network Analysis Corporation, Sixth Semiannual Technical Report for the Project: Local, Regional and Large Scale Integrated Networks, Volume 4, February 1976.

[Fraser 72] Fraser, A. G., "On the interface between computers and data communication systems," *Communications of the ACM* 15 (7), July 1972, 566-573.

[Fraser 74] Fraser, A. G., "SPIDER - An experimental data communication system," in *Pr eedings of the International Conference on Data Communication*, Minneapolis, IEEE, June 1974.

[Fraser 75] Fraser, A. G., "A virtual channel network," *Datamation* 21 (2), February 1975, 51-56.

[Fuchel 75] Fuchel, K., and S. Heller, *Two Dissimilar Networks--Is Marriage Possible?* Brookhaven National Laboratory, Upton, New York, Technical Report BNL-20196, 1975.

[Fuller 76] Fuller, S. H., "Price/performance comparison of C.mmp and the PDP-10," in *Proceedings of the Third Annual Symposium on Computer Architecture*, as *ACM SIGARCH* 4 (4), January 1976, pp. 195-202.

[Fuller 78] Fuller, S. H., et al., "Multi-microprocessors: An overview and working example," *Proceedings of the IEEE* 66 (2), February 1978, 216-228.

[Gerhardstein 78] Gerhardstein, L. H., J. O. Schroeder, and A. J. Boland, "The Pacific Northwest Laboratory minicomputer network," in *Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks*, San Francisco, August 1978.

[Giertz 78] Giertz, H. W., V. Vucins, and L. Ingre, "Experimental fiber optic databus," in *Proceedings of the Fourth ECOC*, Genoa, September 1978, pp. 641-645.

[Gifford 79a] Gifford, D. K., "Weighted voting for replicated data," in *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 150-162.

[Gifford 79b] Gifford, D. K., *Violet, an Experimental Decentralized System*, Xerox PARC Technical Report CSL-79-12, September 1979.

[Gord 73] Gord, E. P., and M. D. Hopwood, *Nonhierarchical Process Structure in a Decentralized Computing Environment*, Technical Report 32, University of California, Irvine, Department of Information and Computer Science, June 1973.

[Gouda 76] Gouda, M. G., and E. G. Manning, "Protocol machines: A concise formal model and its automatic implementation," in *Proceedings of tne Third International Conference on Computer Communication*, 1976.

[Green 77] Green, M. I., "A DoD local network - A structured implementation," in *Proceedings of the Conference on a Second Look at Local Computer Networking*, University of Minnesota, October 1977.

[Grosch 53] Grosch, H. R. J., "High speed arithmetic: The digital computer as a research tool," *Journal of the Optical Society of America* 43 (4), April 1953, 306-310.

[Hafner 74a] Hafner, E. R., "Digital communication loops--A survey," in *Proceedings of the 1974 Zurich International Seminar on Digital Communications*, p. D1.

[Hafner 74b] Hafner, E. R., et al., "A digital loop communication system," *IEEE Transactions on Communications* COM-22 (6), June 1974, 877.

[Hafner 76] Hafner, E. R., and Z. Nenadal, "Enhancing the availability of a loop system by meshing," in *Proceedings of the 1976 International Zurich Seminar on Digital Communications*, pp. D4.1-D4.5

[Harriman 77] Harriman, E., "A microprocessor based implementation of a data stream protocol processor," M.I.T. Laboratory for Computer Science, Local Network Note No. 8, June 15, 1977.

[Hassing 73] Hassing, T. E., R. M. Hampton, G. W. Bailey, and R. S. Gardella, "A loop network for general purpose data communication in a heterogeneous world," in *Proceedings of the Third Data Communications Symposium*, November 1973, pp. 88-96.

[Hayes 71a] Hayes, J. F., and D. N. Sherman, "Traffic analysis of a ring switched data transmission system," *Bell System Technical Journal* 50 (9), November 1971, 2947-2978.

[Hayes 71b] Hayes, J. F., and D. N. Sherman, "Traffic and delay in a circular data network," in *Proceedings of the Second Symposium on Problems in the Optimization of Data Communication Systems*, October 1971, pp. 102-107.

[Heart 73] Heart, F. E., S. M. Ornstein, W. R. Crowther, and W. B. Barker, "A new minicomputer/multiprocessor for the ARPA network," in *AFIPS Conference Proceedings, Volume 42: National Computer Conference*, New York, June 1973, pp. 529-537.

[Hertzberg 79] Hertzberg, R. Y., J. R. Shultz, and J. F. Wanner, "The PROMIS network," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 87-110.

[Hewitt 75] Hewitt, C. E., and B. Smith, "Towards a programming apprentice," *IEEE Transactions on Software Engineering* SE-1 (1), March 1975, 26-45.

[Hill 79] Hill, R., "A table driven approach to cyclic redundancy check calculations," *ACM Computer Communication Review* 9 (2), April 1979, 40-60.

[Holmgren 79] Holmgren, S. F., A. P. Skelton, and D. A. Gomberg, *FY79 Final Report: Cable Bus Applications in Command Centers*, MITRE Technical Report MTR-79W00383, October 1979.

[Hopkins 79] Hopkins, G.T., "Multimode communication on the MITRENET," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 169-177.

[Hopper 77] Hopper, A., "Data ring at Computer Laboratory, University of Cambridge," in *Computer Science and Technology: Local Area Networking*, National Bureau of Standards, NBS Special Publication 500-31, Washington, D.C., August 22-23, 1977, pp. 11-16.

[Hopper 78] Hopper, A., *Local Area Computer Communication Networks*, Ph.D. thesis, University of Cambridge, Computer Laboratory, April 1978.

[Hopper 79] Hopper, A., and D. Wheeler, "Maintenance of ring communication systems," *IEEE Transactions on Communication* COM-27 (4), April 1979, 760-761.

[HP 80] Hewlett-Packard, advertisement in *Electronics*, May 22, 1980.

[Huynh 76] Huynh, D., H. Kobayashi, and F. F. Kuo, "Design issues for mixed media packet switching networks," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 541-549.

[IBM 74] "IBM Synchronous Data Link Control General Information", GA27-3093-0, File No. GENL-09, IBM Systems Development Division, Publications Center, North Carolina, 1974.

[IEEE 75] *IEEE Standard Digital Interface for Programmable Instrumentation*, IEEE Standard 488-1975, IEEE Instrumentation and Measurements Group, October 1975.

[IEN-111 79] Postel, J., "Internet Protocol," IEN 111, USC/Information Sciences Institute, August 1979.

[IEN-112 79] Postel, J., "Transmission Control Protocol," IEN 112, USC/Information Sciences Institute, August 1979.

[Innes 75] Innes, D. R., and J. L. Alty, "An intra university network," in *Proceedings of the Fourth Data Communications Symposium*, October 1975, pp. 1-8 - 1-13.

[Jensen 78] Jensen, E. D., "The Honeywell experimental distributed processor--An overview," *IEEE Computer* 11 (1), January 1978, 28-38.

[Jones 78] Jones, A. K., et al., "Programming issues raised by a multiprocessor," in *Proceedings of the IEEE* 66 (2), February 1978, 229-237.

[Kahn 75] Kahn, R. E., "The organization of computer resources into a packet radio network," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 177-186.

[Katzman 78] Katzman, J. A., "A fault-tolerant computing system," in *Proceedings of the Eleventh Hawaiian International Conference on System Sciences*, 1978, pp. 85-102.

[Kelso 77] Kelso, H. F., *Overview of LASL Integrated Computer Network*, Los Alamos Scientific Laboratory of the University of California, LA-6756-MS, Los Alamos, New Mexico, January 1977.

[Kent 77] Kent, S. T., "Some thoughts on TCP and communication security," M.I.T. Laboratory for Computer Science, Local Network Note No. 6, April 26, 1977.

[Kleinrock 75] Kleinrock, L., and F. Tobagi, "Random access techniques for data transmission over packet switched radio channels," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 187-202.

[Kleinrock 78] Kleinrock, L., "Principles and lessons in packet communications," in *Proceedings of the IEEE* 66 (11), November 1978, pp. 1320-1329.

[Kropfl 72] Kropfl, W. J., "An experimental data block switching system," *Bell System Technical Journal* 51 (6), July-August 1972, 1147-1165.

[Kuo 73] Kuo, F. F., and N. Abramson, "Some advances in radio communication for computers," in *Proceedings Compcon Spring 73*, February 1973, pp. 57-60.

[Kuo 75] Kuo, F. F., and D. R. Binder, "Computer-communication by radio and satellite: The ALOHA system," in R. L. Grimsdale and F. F. Kuo (eds.), *Computer Communication Networks*, NATO Advanced Studies Institute Series, E-4, Noordhoff, Leyden, The Netherlands, 1975, pp. 397-408.

[Labonte 73] Labonte, R. C., "A general purpose digital communication system for operation on a conventional CATV system," in *Proceedings Compcon Spring 73*, February 1973, pp. 85-88.

[Lampson 74] Lampson, B. W., "Redundancy and robustness in memory protection," in *Information Processing 74: Proceedings of the IFIP Congress*, 1974, pp. 18-24.

[Lampson 76] Lampson, B. W., and H. E. Sturgis, "Reflections on an operating system design," *Communications of the ACM* 19 (5), May 1976, 251-265.

[Lederberg 78] Lederberg, J., "Digital communications and the conduct of science: The new literacy," *Proceedings of the IEEE* 66 (11), November 1978, 1314-1319.

[Levine 82] Levine, R. D., "Supercomputers," *Scientific American* 246 (1), January 1982, 118-135.

[Lidinsky 76] Lidinsky, W. P., "The Argonne intra-laboratory network," in *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence Berkeley Laboratory, May 1976, pp. 263-275.

AD-A125 441 COMMUNICATION ENVIRONMENTS FOR LOCAL NETWORKS(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST P V MOCKAPETRIS DEC 82 ISI/RR-82-103 MDA903-81-C-0335

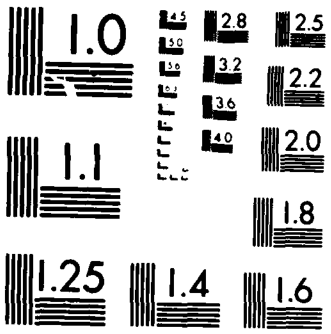
3/3

UNCLASSIFIED F/G 17/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963 A

[Lin 78] Lin, K., "Design of a packet-switched micro-subnetwork," in *Proceed, ,s Compcon Fall 78*, Washington, D. C., September 1978, pp. 184-193.

[Liu 77] Liu, M. T., and C. C. Reames, "Message communication protocol and operating system design for the Distributed Loop Computer Network (DLCN)," in *Proceedings of the Fourth Annual Symposium on Computer Architecture*, March 1977, pp. 193-200.

[Liu 78] Liu, M. T., "Distributed Loop Computer Networks," in *Advances in Computers*, Vol. 17, Academic Press, 1978, pp. 163-221.

[Livesey 79] Livesey, J., "Inter-process communication and naming in the Mininet System," in *Proceedings Compcon Spring 79*, San Francisco, February-March 1979, pp. 222-229.

[Lockheed 72] *SUE Computer Handbook*, Lockheed Electronics Company, 1972.

[Loomis 73] Loomis, D. C., "Ring communication protocols," University of California, Irvine, Department of Information and Computer Science, Technical Report 26, January 1973.

[Luczak 78] Luczak, E. C., "Global bus computer communication techniques," in *Proceedings of the Computer Networking Symposium*, Gaithersburg, Md., December 1978, pp. 58-71.

[Lynch 68] Lynch, W. C., "Reliable full-duplex file transmission over half-duplex telephone lines," *Communications of the ACM* 11 (6), June 1968, 407-410.

[Lynn 82] Lynn, C., personal communication, May 18, 1982.

[Manning 74] Manning, E. G., and R. W. Peebles, *A Homogeneous Network for Data Sharing Communication*, University of Waterloo, Computer Communication Network Group, Technical Report CCNG-E-12, March 1974.

[Manning 77] Manning, E. G., and R. W. Peebles, "A homogeneous network for data sharing," *Computer Networks* 1, 1977, 211-224.

[Mathison 78] Mathison, S. L., "Commercial, legal, and international aspects of packet communications," *Proceedings of the IEEE* 66 (11), November 1978, 1527-1539.

[McMahon 78] McMahon, T. F., "A contention ring implementation of a local data network," M.I.T. Laboratory for Computer Science, Local Network Note No. 17, February 28, 1978.

[McQuillan 73] McQuillan, J. M., *Throughput in the ARPA network - Analysis and measurement*, Bolt Beranek and Newman Inc., Technical Report 2491, Cambridge, Mass., January 1973.

[McQuillan 77] McQuillan, J. M., and D. C. Walden, "The ARPA network design decisions," *Computer Networks* 1 (5), September 1977, 243-289.

[McQuillan 78] McQuillan, J. M., "Enhanced message addressing capabilities for computer networks," *Proceedings of the IEEE* 66 (11), November 1978, 1517-1527.

[McQuillan 79] McQuillan, J. M., "Local network technology and the lessons of history," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 191-197.

[McQuillan 80] McQuillan, J. M., I. Richer, and E. C. Rosen, "The new routing algorithm for the ARPANET," *IEEE Transactions on Communications* COM-28 (5), May 1980, 711-719.

[Meisner 77] Meisner, N. B., et al., "Time division digital bus techniques implemented on coaxial cable," in *Proceedings of the Computer Networking Symposium*, National Bureau of Standards, Gaithersburg, Md., December 1977.

[Metcalfe 73] Metcalfe, R. M., *Packet Communication*," M.I.T. Project MAC Technical Report TR-114, December 1973.

[Metcalfe 76] Metcalfe, R. M., and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM* 19 (7), July 1976, 395-404. Appeared in an earlier version as Xerox PARC Technical Report CSL-75-7, November 1975.

[Metcalfe 77] Metcalfe, R. M., D. R. Boggs, C. P. Thacker, and B. W. Lampson, "Multipoint data communication system with collision detection," United States Patent 4,063,220, December 1977.

[Mills 76] Mills, D. L., "An overview of the distributed computer network," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 523-531.

[Mockapetris 77] Mockapetris, P. V., M. R. Lyle, and D. J. Farber, "On the design of local network interfaces," in *Information Processing 77: Proceedings of the IFIP Congress*, Toronto, August 1977, pp. 427-430.

[Mockapetris 78] Mockapetris, P. V., *Design Considerations and Implementation of the ARPA LNI Name Table*, University of California, Irvine, Department of Information and Computer Science, Technical Report No. 92, April 1978.

[Mockapetris 80] Mockapetris, P. V., and D. J. Farber, *Experiences with the Distributed Computer System*, University of California, Irvine, Department of Information and Computer Science, Technical Report 116, 1980.

[Myers 80] Myers, G. J., *Digital System Design with LSI Bit-Slice Logic*, Wiley-Interscience, 1980.

[Needham 78] Needham, R. M., and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM* 21 (12), December 1978, 993-999.

[Nessett 78] Nessett, D. M., "Protocols for buffer-space allocation in CSMA broadcast networks with intelligent interfaces," in *Proceedings of the Third Conference on Local Computer Networks*, University of Minnesota, October 1978.

[NIC47000 79] *ARPANET Directory*, ARPANET Network Information Center, December 1979.

[NIC7104 76] *ARPANET Protocol Handbook*, ARPANET Network Information Center, April 1976.

[Okuda 78] Okuda, N., T. Kunikyo, and T. Kaji, "Ring century bus - An experimental high speed channel for computer communication," in *Proceedings of the Fourth International Conference on Computer Communication*, Kyoto, September 1978, pp. 161-166.

[Pardo 79] Pardo, R., and M. T. Liu, "Multidestination protocols for distributed systems," in *Proceedings of the Computer Networking Symposium*, National Bureau of Standards, Gaithersburg, Md., 1979, pp. 176-185.

[Peterson 79] Peterson, J. L., "Notes on a workshop on distributed computing," *ACM Operating Systems Review* 13 (3), July 1979, 18-27.

[Petri 62] Petri, C. A., "Kommunikation mit Automaten" (Communication with Automata), Schriften des Rheinisch - Westfalischen Institut fur Instrumentelle Mathematik an der Universitat Bonn, Hft. 2, Bonn, 1962. (Translated in M.I.T. Project MAC Memorandum MAC-M-212.)

[Pickens 79] Pickens, J. R., E. J. Feinler, and J. E. Mathis, "The NIC name server--A datagram based information utility," ARPA Network RFC 756, USC/Information Sciences Institute, July 1979.

[Pierce 72a] Pierce, J. R., "How far can data loops go?" *IEEE Transactions on Communication* COM-20 (3), June 1972, 527-530.

[Pierce 72b] Pierce, J. R., "Network for block switching of data," *Bell System Technical Journal* 51 (6), July-August 1972, 1133-1143.

[Plummer 78] Plummer, W., "Sequence number arithmetic", IEN 74, USC/Information Sciences Institute, September 1978.

[Pogran 76a] Pogran, K. T., "Introducing 'Local Network Notes'," M.I.T. Laboratory for Computer Science, Local Network Note No. 1, November 15, 1976.

[Pogran 76b] Pogran, K. T., and D. D. Clark, "Timeline for implementation of the LCS network," M.I.T. Laboratory for Computer Science, Local Network Note No. 4, December 17, 1976.

[Pogran 76c] Pogran, K. T., "Plan for a local, high-speed computer network for the Laboratory for Computer Science," M.I.T. Laboratory for Computer Science, Local Network Note No. 2, November 12, 1976.

[Pogran 77] Pogran, K. T., "A brief overview of the LCS network," M.I.T. Laboratory for Computer Science, Local Network Note No. 12, September 15, 1977.

[Postel 74] Postel, J., *Survey of Network Control Programs in the ARPA Computer Network,* The MITRE Corporation, Technical Report MTR-6722, McClean, Va., June 1974.

[Postel 81a] Postel, J. (ed.), "Transmission Control Protocol - DARPA internet program protocol specification," ARPA Network RFC 793, USC/Information Sciences Institute, September 1981.

[Postel 81b] Postel, J. (ed.), "Internet Protocol - DARPA internet program protocol specification," ARPA Network RFC 791, USC/Information Sciences Institute, September 1981.

[Postel 81c] Postel, J., "Assigned numbers," ARPA Network RFC 790, USC/Information Sciences Institute, September 1981.

[Postel 81d] Postel, J., "NCP/TCP transition plan," ARPA Network RFC 801, USC/Information Sciences Institute, November 1981.

[Postel 81e] Postel, J., "Internet meeting notes--28-29-30 January 1981," IEN 175, USC/Information Sciences Institute, March 1981.

[Pouzin 76] Pouzin, L., "Virtual circuits vs. datagrams - Technical and political problems," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 483-494.

[Ratliff 78] Ratliff, S. L., "A Dynamic Routing Algorithm for a Local Packet Network," B.S. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1978.

[Rawson 78] Rawson, E. G., and R. M. Metcalfe, "Fibernet: Multimode optical fibers for local computer networks," *IEEE Transactions on Communication* COM-26 (7), July 1978, 983-990.

[Rawson 79] Rawson, E. G., "Application of fiber optics to local networks," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 155-168.

[Reames 75] Reames, C. C., and M. T. Liu, "A loop network for simultaneous transmission of variable-length messages," in *Proceedings of the Second Annual Symposium on Computer Architecture*, Houston, Texas, 1975, pp. 7-12.

[Reames 76] Reames, C. C., and M. T. Liu, "Design and simulation of the Distributed Loop Computer Network (DLCN)," in *Proceedings of the Third Annual Symposium on Computer Architecture*, January 1976, pp. 124-129.

[Redell 74] Redell, D. D., *Naming and Protection in Extensible Operating Systems*, Ph.D. thesis, Massachusetts Institute of Technology, November 1974. (Also M.I.T. Project MAC TR-140.)

[Reed 76] Reed, D. P., "Protocols for the network," M.I.T. Laboratory for Computer Science, Local Network Note No. 3, November 29, 1976.

[Reed 77] Reed, D. P., "The initial connection mechanism in DSP," M.I.T. Laboratory for Computer Science, Local Network Note No. 10, September 15, 1977.

[Retz 75] Retz, D. L., "Operating system design considerations for the packet-switching environment," in *AFIPS Conference Proceedings, Volume 44: National Computer Conference*, Anaheim, Calif., May 1975, pp. 155-160.

[Retz 76] Retz, D. L., and B. W. Schafer, "Structure of the ELF operating system," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 1007-1016.

[Richardson 79] Richardson, T. G., and L. W. Yu, "The effect of protocol on the response time of loop structures for data communications," *Computer Networks* 3 (1), February 1979, 57-66.

[Ritchie 74] Ritchie, D. M., and K. Thompson, "The UNIX timesharing system," *Communications of the ACM* 17 (7), July 1974, 365-375.

[Roberts 67] Roberts, L. G., "Multiple computer networks and intercomputer communication," ACM Symposium on Operating Systems Problems, October 1967.

[Roberts 70] Roberts, L. G., and B. D. Wessler, "Computer network development to achieve resource sharing," in *AFIPS Conference Proceedings, Volume 36: Spring Joint Computer Conference*, Atlantic City, N. J., May 1970, pp. 543-549.

[Roberts 74] Roberts, L. G., "Dynamic allocation of satellite capacity through packet reservation," in *AFIPS Conference Proceedings, Volume 43: National Computer Conference*, Chicago, June 1974, pp. 711-716.

[Roberts 78] Roberts, L. G., "The evolution of packet switching," in *Proceedings of the IEEE* 66 (11), November 1978, 1307-1313.

[Rodgers 77] Rodgers, J. C., "Computer networking with a data bus," in *Proceedings of the Sixteenth Annual Technical Symposium*, Association for Computing Machinery (Washington, D.C. chapter) and the National Bureau of Standards, June 1977, pp. 45-50.

[Rosen 73] Rosen, S., and J. M. Steele, "A local computer network," in *Proceedings Compcon Spring 73*, San Francisco, February 1973, pp. 129-132.

[Rothnie 77] Rothnie, J. B., N. Goodman, and P. A. Bernstein, *The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case)*, Computer Corporation of America, Report No. CCA-77-02, 1977.

[Rovner 78] Rovner, P. D., "Message flow control in a local network," University of Rochester, Computer Science Department, Rochester N.Y., 1978.

[Rowe 73] Rowe, L. A., M. D. Hopwood, and D. J. Farber, "Software methods for achieving fail-soft behavior in the distributed computing system," in *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability*, May 1973, pp. 7-11.

[Rowe 75] Rowe, L. A., *The Distributed Computing Operating System*, University of California, Irvine, Department of Information and Computer Science, Technical Report 66, 1975.

[Rowe 79] Rowe, L. A., and K. P. Birman, "Network support for a distributed data base system," in *Proceedings of the Fourth Berkeley Conference on Distributed Data Bases and Computer Networks*, Lawrence Berkeley Laboratory, August 1979, pp. 337-352.

[RS232C 69] "Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange," Electronics Industries Association, August 1969.

[Ryan 81] Ryan, R., et al., "Intel local network architecture," *IEEE Micro*, November 1981, pp. 26-41.

[Sakai 77] Sakai, T., T. Hayashi, S. Kitazawa, K. Tabata, and T. Kanade, "Inhouse Computer Network KUIPNET," in *Information Processing 77: Proceedings of the IFIP Congress*, August 1977.

[Saltzer 79] Saltzer, J. H., and K. T. Pogran, "A star-shaped ring network with high maintainability," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 179-190.

[Schantz 74] Shantz, R. E., *Operating System Design for a Network Computer*, State University of New York at Stony Brook, Department of Computer Science, Technical Report 28, May 1974.

[Sherman 77] Sherman, R. H., et al., "Current summary of Ford activities in local networking," in *Local Area Networking*, NBS Special Publication 500-31, 1977.

[Sherman 78] Sherman, R. H., M. E. Gable, and G. McClure, "Concepts, strategies for local data network architectures," *Data Communications*, 7 (7), July 1978, 39-49.

[Shoch 79a]	Shoch, J. F., *Design and Performance of Local Computer Networks*, Ph.D. thesis, Stanford University, August 1979.
[Shoch 79b]	Shoch, J. F., and J. A. Hupp, "Performance of an Ethernet local network: A preliminary report," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 113-125.
[Smith 79]	Smith, S. M., "Use of broadband coaxial cable networks in an assembly plant environment," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 67-74.
[Smith 80a]	Smith, K., "Coax car system replaces harness," *Electronics*, August 28, 1980, 74-76.
[Smith 80b]	Smith, K., "Chips, twisted pair build simple local net," *Electronics*, August 28, 1980, 80.
[Spragins 71]	Spragins, J. D., "Analysis of loop transmission systems," in *Proceedings of the Second Symposium on Problems in the Optimization of Data Communication Systems*, October 1971, pp. 175-182.
[Stonebreaker 79]	Stonebreaker, M., "Concurrency control and consistency of multiple copies of data in distributed INGRESS," *IEEE Transactions on Software Engineering* SE-5 (3), May 1979, 188-194.
[Svobodova 77a]	Svobodova, L., "Comparative study between the Ethernet and the Ringnet," M.I.T. Laboratory for Computer Science, Local Network Note No. 13, September 27, 1977.
[Svobodova 77b]	Svobodova, L., "Models for performance studies of the Ethernet and the Ringnet," M.I.T. Laboratory for Computer Science, Local Network Note No. 14, October 14, 1977.
[Svobodova 77c]	Svobodova, L., "Evaluation of transmission protocols for the Ethernet," M.I.T. Laboratory for Computer Science, Local Network Note No. 16, October 14, 1977.
[Swinehart 79]	Swinehart, D., G. McDaniel, and D. Boggs, *WFS: A Simple Shared File System for a Distributed Environment*, Xerox PARC Technical Report CSL-79-13. Also in *ACM Operating Systems Review* 13 (5), November 1979.
[Switzer 72]	Switzer, I., "The cable television system as a computer-communication network," in *Proceedings of the Symposium on Computer Communications Networks and Teletraffic*, Jerome Fox, ed., Polytechnic Press, New York, April 1972, pp. 339-346.
[Szurkowski 78]	Szurkowski, E., "A high bandwidth local computer network," *Proceedings Compcon Fall 78*, Washington, D. C., September 1978, pp. 98-103.

[Tang 76] Tang, C. K., "Cache system design in the tightly coupled multiprocessor system," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 749-753.

[Thacker 79] Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, *ALTO: A Personal Computer*, Xerox PARC Technical Report CSL-79-11, August 1978.

[Thomas 73] Thomas, R. H., "A resource sharing executive for the ARPANET," in *AFIPS Conference Proceedings, Volume 42: National Computer Conference*, New York, June 1973, pp. 155-163.

[Thomas 79] Thomas, R. H., "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems* 4 (2), June 1979, 180-209.

[Thornton 75] Thornton, J. E., G. S. Christensen, and P. D. Jones, "A new approach to network storage management," *Computer Design* 14 (11), November 1975, 81-85.

[Thornton 79] Thornton, J. E., "Overview of the HYPERchannel," in *Proceedings Compcon Spring 79*, San Francisco, February 1979, pp. 262-265.

[Tobagi 76] Tobagi, F. A., S. E. Lieberson, and L. Kleinrock, "On measurement facilities in packet radio systems," in *AFIPS Conference Proceedings, Volume 45: National Computer Conference*, New York, June 1976, pp. 589-596.

[Todd 78] Todd, S., "Algorithm and hardware for a merge sort using multiple processors," *IBM Journal of Research and Development* 22 (5). September 1978, 509-517.

[Tokoro 77] Tokoro, M., and K. Tamaru, "Acknowledging Ethernet," in *Proceedings Compcon Fall 77*, Washington, D. C., September 1977, pp. 320-325.

[Tolmie 76] Tolmie, D. E., "The LASL integrated computer network," in *Proceedings of the University of Minnesota Conference on Local Networks*, September 1976.

[Tymes 71] Tymes, L. R., "TYMNET - A terminal oriented communication network," in *AFIPS Conference Proceedings, Volume 38: Spring Joint Computer Conference*, Atlantic City, N. J., May 1971, pp. 211-216.

[Urbina 78] Urbina, A., "Performance of a terminal concentrator under a data stream protocol," M.I.T. Laboratory for Computer Science, Local Network Note No. 18, March 27, 1978.

[Vittal 73] Vittal, J., and D. J. Farber, "On the interfacing of multiple DCS networks and the ARPANET," DCS memo 62, University of California, Irvine, Department of Information and Computer Science, March 1973.

[Watson 77] Watson, R. W., and J. G. Fletcher, "End-to-end protocol design issues for a local computer network: The Delta T mechanism," in *Proceedings of the Conference on a Second Look at Local Networking*, University of Minnesota, October 1977.

[Watson 78] Watson, R. W. "The LLL Octopus network: Some lessons and future directions," in *Proceedings of the Third USA-Japan Computer Conference*, San Francisco, October 1978.

[Watson 79] Watson, R. W., and J. G. Fletcher, "An architecture for support of network operating system services," in *Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks*, Lawrence Berkeley Laboratory, August 1979, pp. 18-50.

[West 72] West, L. P., "Loop transmission control structures," *IEEE Transactions on Communication* COM-20 (3), June 1972, 531-539.

[West 77] West, A. R., A broadcast packet-switched computer communication network--Design progress report," Queen Mary College (University of London), Department of Computer Science and Statistics, unpublished report, February 1977.

[Wilkes 75] Wilkes, M. V., "Communication using a digital ring," in *Proceedings of the Pacific Area Computer Communication Network System Symposium*, Sendai, Japan, August 1975, pp. 47-56.

[Wilkes 79] Wilkes, M. V., and D. J. Wheeler, "The Cambridge digital communication ring," in *Proceedings of the Local Area Communications Network Symposium*, Boston, May 1979, pp. 47-61.

[Wilkes 80] Wilkes, M. V., and R. M. Needham, "The Cambridge model distributed system," *ACM Operating Systems Review* 14 (1), January 1980.

[Willard 74] Willard, D. G., "A time division multiple access system for digital communication," *Computer Design* 13 (6), June 1974, 79-83.

[Wittie 76] Wittie, L., "Efficient message routing in mega-micro-computer networks," in *Proceedings of the Third Annual Symposium on Computer Architecture*, as *ACM SIGARCH* 4 (4), January 1976.

[Wulf 72] Wulf, W. A., and C. G. Bell, "C.mmp - A multi-mini-processor," in *AFIPS Conference Proceedings, Volume 41: Fall Joint Computer Conference*, Part II, Anaheim, Calif., December 1972, pp. 765-777.

[Wulf 74] Wulf, W. A., et al., "HYDRA: The kernel of a multiprocessor operating system," *Communications of the ACM* 17 (6), June 1974, 337-345.

[Wulf 78] Wulf, W. A., and S. P. Harbison, "Reflections in a pool of processors: An experience report on C.mmp/Hydra," in *AFIPS Conference Proceedings, Volume 47: National Computer Conference*, Anaheim, Calif., June 1978, pp. 939-951.

[Xerox 81a] *Internet Transport Protocols*, Xerox System Integration Standard, Xerox Corporation, Stamford, Conn., December 1981.

[Xerox 81b] *Courier: The Remote Procedure Call Protocol*, Xerox System Integration Standard, Xerox Corporation, Stamford, Conn., December 1981.

[Yajima 77] Yajima, S., Y. Kambayashi, S. Yoshida, and K. Iwama, "Labolink: An optically linked laboratory computer network," *IEEE Computer* 10 (11), November 1977, 52-59.

[Yatsuboshi 78] Yatsuboshi, R., T. Tsuda, K. Yamaguchi, and Y. Inoue, "An in-house network configuration for distributed intelligence," in *Proceedings of the Fourth International Conference on Computer Communication*, Kyoto, September 1978, pp. 155-160.

[Yuill 76] Yuill, S. J., "A survey of multiple access satellite packet communication," in *Proceedings of the Fifteenth Annual Technical Symposium*, Association for Computing Machinery (Washington, D.C. chapter) and the National Bureau of Standards, June 1976, pp. 9-16.

[Zafiropulo 72] Zafiropulo, P., and E. H. Rothauser, "Signalling and frame structures in highly decentralized loop systems," in *Proceedings of the First International Conference on Computer Communication*, Washington, D. C., October 1972, pp. 309-315.

[Zafiropulo 74] Zafiropulo, P., "Performance evaluation of reliability improvement techniques for single-loop communication systems," *IEEE Transactions on Communication* COM-22 (6), June 1974, 742-751.

4-8

DTI